Graph-XLL: a Graph Library for Extra Large Graph Analytics on a Single Machine

by

Jian Wu
B.Eng., Wuhan University, 2010
M.A.Sc., University of Chinese Academy of Sciences, 2013
Ph.D., University of Victoria, 2017

A Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

in the Department of Computer Science

Graph-XLL: a Graph Library for Extra Large Graph Analytics on a Single Machine

by

Jian Wu
B.Eng., Wuhan University, 2010
M.A.Sc., University of Chinese Academy of Sciences, 2013
Ph.D., University of Victoria, 2017

Supervisory Committee

Dr. Alex Thomo, Co-Supervisor
(Department of Computer Science)

Dr. Venkatesh Srinivasan, Co-Supervisor
(Department of Computer Science)

Supervisory Committee

Dr. Alex Thomo, Co-Supervisor
(Department of Computer Science)

Dr. Venkatesh Srinivasan, Co-Supervisor
(Department of Computer Science)

ABSTRACT

Graph libraries containing already-implemented algorithms are highly desired since users can conveniently use the algorithms off-the-shelf to achieve fast analytics and prototyping, rather than implementing the algorithms with lower-level APIs. Besides the ease of use, the ability to efficiently process extra large graphs is also required by users. The popular existing graph libraries include the igraph R library and the NetworkX Python library. Although these libraries provide many off-the-shelf algorithms for users, the in-memory graph representation limits their scalability for computing on large graphs. Therefore, in this work, we develop Graph-XLL: a graph library implemented using the WebGraph framework in a vertex-centric manner, with much less memory requirement compared to igraph and NetworkX. Scalable analytics for extra large graphs (up to tens of millions of vertices and billions of edges) can be achieved on a single consumer grade machine within a reasonable amount of time. Such computation would cause out-of-memory error if using igraph or NetworkX.

Contents

List of Tables

List of Figures

ACKNOWLEDGEMENTS

I would like to express my great gratitude to:

my supervisors, Dr. Alex Thomo, and Dr. Venkatesh Srinivasan, for their meticulous supervision on my Master's thesis, their generosity of supporting my research, and their inspirational leadership, aptitude, and enthusiasm for scientific research;

my colleagues, Fatemeh Esfahani, Yudi Santoso, and Diana Popova, for providing insightful discussions and assistance to my research;

my parents, for their unconditional love and support all along.

DEDICATION

To my parents,
and
everyone who offered the help
along the way.

Chapter 1

Introduction

1.1 Motivation

Graph analytics are becoming increasingly important since graphs are a proper abstraction for complex systems and thus can be used in many areas such as social network analysis, neural network analysis, public transportation routing, epidemiology [1–4], etc. Notably, the Best-Paper-Award of VLDB 2018 was given to the work of Sahu et al. [5], which conducted a thorough study of the needs of industry practitioners working with graph data. Some of their most important findings, which motivate our work, were as follows:

1. Many graphs are quite large, often containing more than a billion edges. Namely, they found that these graphs represent an enormously wide range of entities and are used by organizations from small businesses to large enterprises. They emphasize that this finding runs counter to a common assumption that large graphs are problematic only for large organizations such as Google, Facebook, and Twitter.

2. The survey also found that scalability is the most pressing challenge faced by users and the ability to process very large graphs efficiently is among the biggest limitation of existing software.

3. The most common request they found was the addition of algorithms that users could use off-the-shelf. Most of software products provide lower-level programming APIs using which users can compose graph algorithms. However, they

found that users of these software products find more value in directly using an already-implemented algorithm than implementing the algorithms themselves.

The igraph R library [6] and the NetworkX Python library [7] are some of the most popular existing graph libraries due to their easy-to-use off-the-shelf feature. Both libraries have implemented important algorithms which can be used with simple function calls. However, they do not scale to large graphs. The main reason for this is their assumption that the graphs and their auxiliary data structures must fit in main memory. This unfortunately is not true for large graphs. Such large graphs cannot be processed by igraph or NetworkX on commodity machines which are ubiquitous among researchers and small to medium businesses.

In this thesis, we develop Graph-XLL (https://graph-xll.github.io), a graph library written in Java, with emphasis on the scalability for extra large graph analytics. To address the large memory footprint issue faced by igraph and NetworkX, we use the WebGraph framework [8] for the underlying graph representation. WebGraph is a highly efficient graph compression framework. Instead of loading the complete graph into the memory, WebGraph stores a memory-mapped compressed graph on the hard drive. Furthermore, in Graph-XLL, we implement the algorithms in a vertex-centric manner. The vertex-centric method performs the graph computation from the perspective of a single vertex and represents graph algorithms as a sequence of iterations, or supersteps [9]. Vertices can be processed independently, such as updating the values by receiving the messages from the previous superstep and "broadcasting" the values or messages for the next superstep. In this computation model the computation can be performed locally and does not require global information. Moreover, vertices can be processed in parallel within each superstep, which can greatly improve the performance.

While a multitude of algorithms have been implemented in Graph-XLL, we focus, in this thesis, on graph centrality measures, diameter and truss-decomposition. Namely, we showcase our implementations for eigenvector, hub, authority, PageRank, betweenness centralities, diameter, and truss-decomposition using the vertex-centric model. Other scalable algorithms in Graph-XLL computing triad-enumeration, core-decomposition, feedback-arc-set, influential-users, and importance-based-communities have been implemented by our previous works [10–16]. There are no algorithms yet implemented for the truss-decomposition, feedback-arc-set, influential-users, and importance-based-communities in igraph or NetworkX despite them being immensely popular concepts in graph analytics. On the other hand, Graph-XLL still misses

a few algorithms for computing cliques and closeness. While igraph and NetworkX have algorithms for them, they are not scalable. The quest for scalable algorithms for computing cliques and closeness is part of our future work.

The contributions of this thesis are summarized as follows:

1. We implement various graph algorithms for centrality analysis (e.g., eigenvector, hub and authority, PageRank and betweenness), diameter, and truss-decomposition with the emphasis on the scalability to achieve extra large graph processing up to tens of millions of vertices and billions of edges.

2. We perform a thorough experimental study to investigate the scalability using different datasets and compare Graph-XLL with igraph and NetworkX in terms of runtime and memory consumption.

3. We prove that Graph-XLL is capable of efficiently analyzing extra large graphs on a single consumer-grade machine.

1.2 Outline

The topic of the thesis is implementing, engineering, and evaluating various important and popular graph algorithms with a focus on scalability.

Chapter 1 introduces the motivation and the outline of the thesis.

Chapter 2 serves as the background chapter, discussing the vertex-centric model, scalability, the WebGraph framework, igraph, and NetworkX.

Chapter 3 details the implementation of graph algorithms for computing centrality measures (eigenvector, hub and authority, PageRank and betweenness), diameter, and truss-decomposition.

Chapter 4 shows the experimental results of using graph algorithms implemented in this work for computing on different datasets ranging from small graphs to extra large graphs with a number of edges up to one billion.

Chapter 5 compares the performance of Graph-XLL with igraph and NetworkX in terms of runtime, memory consumption, and scalability.

Chapter 6 summarizes the thesis and discusses the future work.

1.3 Publications

The publications during the Master's program are listed below:

1. "Graph-XLL: a Graph Library for Extra Large Graph Analytics on a Single Machine", Jian Wu, Venkatesh Srinivasan, and Alex Thomo, IISA 2019.

2. "K-Truss Decomposition of Large Networks on a Single Consumer-Grade Machine", Jian Wu, Alison Goshulak, Venkatesh Srinivasan, and Alex Thomo, ASONAM 2018.

3. "Fast Truss Decomposition in Large-scale Probabilistic Graphs", Fatemeh Esfahani, Jian Wu, Venkatesh Srinivasan, Alex Thomo, and Kui Wu, EDBT 2019.

Chapter 2

Background

In this chapter, we introduce the concepts of the vertex-centric model and scalability, the WebGraph framework, the igraph library and the NetworkX library.

2.1 Vertex-Centric Model

The vertex-centric model, as its name suggests, is a programming model for graph processing which is centered around the vertices [9]. The vertex-centric model performs the graph computation from the perspective of a single vertex (some people call it "thinking like a vertex") and represents graph algorithms as a sequence of iterations, or supersteps. Vertices are processed independently by a vertex program, such as updating the values by receiving the messages coming from its neighbors from the previous superstep and "broadcasting" the values or messages to its neighbors for the next superstep. For each vertex, the only information it has is its neighbor list and its own properties. A vertex-centric program runs iteratively. In each iteration, the vertex program is executed by each vertex and messages are exchanged between vertices. The program stops when there are no messages sent from any vertex.

There are many advantages using the vertex-centric programming model for graph processing. The main advantage is that it becomes very easy to parallelize the algorithm using the vertex-centric model. In each iteration, each vertex executes the vertex program independently. Therefore, in each iteration, the vertex program can be executed in parallel for different vertices. Besides the ease to parallelize, the vertex-centric model is also suitable for distributed systems. It makes implementing a distributed graph algorithm much easier and simpler.

2.2 Scalability

Scalability of a program can be defined in two ways: the ability to handle increased workload without adding resources to a system; the ability to handle increased workload by repeatedly applying a cost-effective strategy for extending a systems capacity [17].

The first definition for scalability assumes that the computing system is a fixed system. An algorithm is classified as scalable if it continues to run properly and accurately as the computation workload increases. The second definition is more complex. The focus is on whether the algorithm can improve the performance or not when, for example, more processors are added. If the algorithm is not able to coordinate work among the added processors properly, eventually, adding more processors will lose the benefit, which will not be cost-effective for extending a system's capacity.

In this thesis, we use the first definition of scalability for an algorithm: the ability to continue to perform properly and accurately as the computation workload increases. We investigate the scalability of an algorithm by increasing the computation workload gradually. We define an algorithm fails to scale if the algorithm is not usable for large computation (e.g., takes too much time), or if the algorithm demands excessive resources (e.g., large memory) to run.

2.3 WebGraph

WebGraph is a highly efficient graph compression framework that allows random access to a memory-mapped compressed graph stored on the hard drive. WebGraph uses lazy techniques that delay the decompression until it is actually necessary when accessing a compressed graph. WebGraph also supports thread-safe operations on an immutable graph, facilitating parallel computation. The documentation and the package can be obtained from the WebGraph home page (http://webgraph.di.unimi.it). For the compression techniques used in WebGraph, please refer to [8]. We choose WebGraph because WebGraph can provide efficient operations (e.g., obtaining the neighbors of a node).

2.4 igraph and NetworkX

The igraph R library [6] and the NetworkX Python library [7] are some of the most popular existing graph libraries due to their easy-to-use off-the-shelf feature. Both libraries use in-memory graph representation, but with different implementations. The igraph library is written in C and provides interfaces for R and Python programming languages. In terms of graph representation, the igraph library uses indexes and vectors for vertices and edges to achieve fast access and iterations over vertices and edges, which is beneficial to the performance. However, such graph data structure is memory consuming and not flexible to make changes to the graph, such as adding or deleting vertices and edges. NetworkX, on the other hand, focuses on the flexibility by using hash tables (dictionaries in Python) as the underlying graph data structure, which would inevitably cause a large memory footprint and a slower speed due to the overhead cost. In short, both libraries have large memory footprints, which limits the scalability to process large graphs efficiently.

2.5 Summary

In this chapter, we briefly introduced the concepts (the vertex-centric model and scalability) and the tools (WebGraph, igraph and NetworkX) used in the thesis. In Graph-XLL, we implement a multitude of graph algorithms using WebGraph in a vertex-centric manner. Scalability is investigated by increasing the computation workload gradually. Transverse comparison in terms of runtime and memory consumption among Graph-XLL, igraph and NetworkX will be made in chapter 5.

Chapter 3

Algorithms Implementation

This chapter describes the graph algorithms computing centrality measures (eigenvector, hub, authority, PageRank and betweenness), diameter, and truss decomposition implemented in Graph-XLL. We implement the algorithms in a vertex-centric manner. To reduce the memory footprint of the algorithms, we use the WebGraph framework, a highly efficient graph compression framework. Computations are broken down to vertex level and are independent between different vertices, facilitating parallel processing.

3.1 Centrality Measures

The centrality measures are used to identify the most important vertices within a graph, including eigenvector, hub, authority, PageRank, betweenness, closeness, etc. Centrality measures are widely used in social network analysis, i.e., to identify the most influential person or people in a social network. In Graph-XLL, we will cover eigenvector, hub, authority, PageRank and betweenness. Other centrality measures (i.e., closeness) will be part of our future work.

3.1.1 Eigenvector Centrality

Eigenvector centrality [18] is the measure of the influence of a vertex in a graph. Relative scores are assigned to all vertices in the graph based on the intuition that a vertex will have high score if it is pointed to by many other vertices with high scores. That is to say, important nodes have important friends. The formal definition of eigenvector centrality is defined as follows.

Eigenvector Centrality (EC) of a vertex v_i in a graph $G = (V, E)$ is defined as

$$EC(v_i) = \frac{1}{\lambda} \sum_{v_j \in IN(v_i)} EC(v_j), \tag{3.1}$$

where $IN(v_i)$ is the set of $v_i's$ in-neighbours (vertices points to v_i) and λ is a constant.

In the matrix form, the eigenvector centrality for all vertices

$$\vec{EC} = [EC(v_1), EC(v_2), \cdots, EC(v_n)] \tag{3.2}$$

can be viewed as the principal left eigenvector of adjacency matrix of the graph. $\vec{EC}$ is the solution to the equation

$$\lambda \vec{EC} = \vec{EC} \cdot \vec{A} = \vec{EC} \begin{bmatrix} a(v_1, v_1) & a(v_1, v_2) & \cdots & a(v_1, v_n) \\ a(v_2, v_1) & \ddots & & \vdots \\ \vdots & & a(v_i, v_j) & \\ a(v_n, v_1) & \cdots & & a(v_n, v_n) \end{bmatrix} \tag{3.3}$$

where λ corresponds to the largest eigenvalue and $\vec{A}$ is the adjacency matrix of the graph: $a(v_i, v_j)$ is 0 if v_i does not link to v_j and 1 if v_i links to v_j.

Algorithm 1 shows the major steps of computing the eigenvector centrality scores. We use two arrays (EC_{prev} and EC_{curr}) to store the eigenvector centrality scores of all vertices for the previous and current supersteps. At the beginning of the algorithm, the scores are initialized to 1 for all vertices. We use MAX_ITER and *tolerance* to control the iteration. MAX_ITER is the maximum value for the superstep. *residual* is the Euclidean distance between EC_{prev} and EC_{curr}, which can be used as the criterion for convergence. In each superstep, each vertex updates its score by summing up the scores of its neighbors pointing to the vertex from the previous superstep, as shown in steps 7 to 11. We normalize the score array by its magnitude. Steps 16 and 17 update EC_{prev} using EC_{curr}, which will be used in the next superstep. The program stops if the *residual* is smaller than the *tolerance* or MAX_ITER is reached.

We implement the program in Java 8 with the WebGraph framework. WebGraph provides APIs which allow random access to a memory-mapped compressed graph stored on a hard drive (step 9), decreasing the main memory footprint significantly. For each vertex, steps 7 to 11 are independent between different vertices. We use Java 8 parallel stream to parallelize steps 7 to 11 for different vertices. EC_{prev} and

Algorithm 1 Eigenvector Centrality Compute Function

```
1: function COMPUTE(G)
2:     if superstep = 0 then
3:         for v ∈ V do
4:             EC_prev[v] ← 1
5:     while superstep < MAX_ITER and residual > tolerance do
6:         superstep ← superstep + 1
7:         for v ∈ V do
8:             sum ← 0
9:             for u ∈ IN(v) do
10:                sum ← sum + EC_prev[u]
11:            EC_curr[v] ← sum
12:        norm ← ‖EC_curr‖
13:        for v ∈ V do
14:            EC_curr[v] ← EC_curr[v]/norm
15:        residual ← ‖E⃗C_curr − E⃗C_prev‖
16:        for v ∈ V do
17:            EC_prev[v] ← EC_curr[v]
```

EC_{curr} are shared among threads. WebGraph can also provide a flyweight copy of the graph for parallelization, which can minimize the overhead memory consumption induced by the parallel process.

3.1.2 Hub Centrality

Hub and authority are two attributes for a vertex introduced by Jon Kleinberg for his work on the Hyperlink-Induced Topic Search (the HITS algorithm) which rates web pages [19]. If a vertex is a hub, it means that it knows where to find information on a given topic. In other works, a good hub represents a vertex which has many links to other vertices. The mathematical definition is defined as follows.

Hub Centrality (HC) of a vertex v_i in a graph $G = (V, E)$ is defined as

$$HC(v_i) = \frac{1}{\lambda} \sum_{v_j \in ON(v_i)} \sum_{v_k \in IN(v_j)} HC(v_k), \tag{3.4}$$

where $ON(v_i)$ is the set of $v_i's$ out-neighbours (vertices with links from v_i), $IN(v_j)$ is the set of $v_j's$ in-neighbours (vertices with links to v_j), and λ is a constant.

Algorithm 2 Hub Centrality Compute Function

```
1: function COMPUTE(G)
2:     if superstep = 0 then
3:         for v ∈ V do
4:             HC_prev[v] ← 1
5:     while superstep < MAX_ITER and residual > tolerance do
6:         superstep ← superstep + 1
7:         /* A⃗^T HC⃗ */
8:         for v ∈ V do
9:             sum ← 0
10:            for u ∈ IN(v) do
11:                sum ← sum + HC_prev[u]
12:            HC_curr[v] ← sum
13:        for v ∈ V do
14:            HC_prev[v] ← HC_curr[v]
15:        /* A⃗A⃗^T HC⃗ */
16:        for v ∈ V do
17:            sum ← 0
18:            for u ∈ ON(v) do
19:                sum ← sum + HC_prev[u]
20:            HC_curr[v] ← sum
21:        norm ← ‖HC_curr‖
22:        for v ∈ V do
23:            HC_curr[v] ← HC_curr[v]/norm
24:        residual ← ‖HC⃗_curr − HC⃗_prev‖
25:        for v ∈ V do
26:            HC_prev[v] ← HC_curr[v]
```

In the matrix form, the hub centrality for all vertices

$$\vec{HC} = [HC(v_1), HC(v_2), \cdots, HC(v_n)]^T$$

can be viewed as the principal right eigenvector of $\vec{A}\vec{A}^T$, where $\vec{A}$ is the adjacency matrix of the graph. $\vec{HC}$ is the solution to the equation

$$\lambda \vec{HC} = \vec{A}\vec{A}^T \cdot \vec{HC}, \tag{3.5}$$

where λ corresponds to the largest eigenvalue.

Algorithm 2 shows the major steps of computing the hub centrality scores. We use

two arrays (HC_{prev} and HC_{curr}) to store the hub centrality scores of all vertices for the previous and current supersteps. At the beginning of the algorithm, the scores are initialized to 1 for all vertices. We use MAX_ITER and *tolerance* to control the iteration. MAX_ITER is the maximum value for the superstep. *residual* is the Euclidean distance between HC_{prev} and HC_{curr}, which can be used as the criterion for convergence. In each superstep, each vertex first updates its score by summing up the scores of its neighbors pointing to the vertex from the previous superstep, as shown in steps 8 to 12. Then each vertex updates its score by summing up the scores of its neighbors which are pointed to by the vertex from the previous superstep. We normalize the score array by its magnitude. Steps 25 and 26 update EC_{prev} using EC_{curr}, which will be used in the next superstep. The program stops if the *residual* is smaller than the *tolerance* or MAX_ITER is reached. We use the WebGraph framework to achieve random access to the compressed graph (steps 10 and 18) and Java 8 parallel stream to parallelize steps 8 to 12 and steps 16 to 20.

3.1.3 Authority Centrality

Authority score [19] is another attribute for a vertex in a graph. It measures how much knowledge, information, etc. held by the vertex on a topic. If a vertex is a good authority, it means that it is linked by many different hubs. The mathematical definition is defined as follows.

Authority Centrality (AC) of a vertex v_i in a graph $G = (V, E)$ is defined as

$$AC(v_i) = \frac{1}{\lambda} \sum_{v_j \in IN(v_i)} \sum_{v_k \in ON(v_j)} AC(v_k), \tag{3.6}$$

where $IN(v_i)$ is the set of $v_i's$ in-neighbours (vertices with links to v_i), $ON(v_j)$ is the set of $v_j's$ out-neighbours (vertices with links from v_j), and λ is a constant.

In the matrix form, the authority centrality for all vertices

$$\vec{AC} = [AC(v_1), AC(v_2), \cdots, AC(v_n)]^T \tag{3.7}$$

can be viewed as the principal right eigenvector of $\vec{A}^T\vec{A}$, where $\vec{A}$ is the adjacency matrix of the graph. $\vec{AC}$ is the solution to the equation

$$\lambda\vec{AC} = \vec{A}^T\vec{A} \cdot \vec{AC}, \tag{3.8}$$

Algorithm 3 Authority Centrality Compute Function

```
1:  function COMPUTE(G)
2:      if superstep = 0 then
3:          for v ∈ V do
4:              AC_prev[v] ← 1
5:      while superstep < MAX_ITER and residual > tolerance do
6:          superstep ← superstep + 1
7:          /* A⃗AC⃗ */
8:          for v ∈ V do
9:              sum ← 0
10:             for u ∈ ON(v) do
11:                 sum ← sum + AC_prev[u]
12:             AC_curr[v] ← sum
13:         for v ∈ V do
14:             AC_prev[v] ← AC_curr[v]
15:         /* A⃗^T A⃗AC⃗ */
16:         for v ∈ V do
17:             sum ← 0
18:             for u ∈ IN(v) do
19:                 sum ← sum + AC_prev[u]
20:             AC_curr[v] ← sum
21:         norm ← ‖AC_curr‖
22:         for v ∈ V do
23:             AC_curr[v] ← AC_curr[v]/norm
24:         residual ← ‖AC⃗_curr − AC⃗_prev‖
25:         for v ∈ V do
26:             AC_prev[v] ← AC_curr[v]
```

where λ corresponds to the largest eigenvalue.

Algorithm 3 shows the major steps of computing the authority centrality scores. We use two arrays (AC_{prev} and AC_{curr}) to store the authority centrality scores of all vertices for the previous and current supersteps. At the beginning of the algorithm, the scores are initialized to 1 for all vertices. We use MAX_ITER and *tolerance* to control the iteration. MAX_ITER is the maximum value for the superstep. *residual* is the Euclidean distance between AC_{prev} and AC_{curr}, which can be used as the criterion for convergence. In each superstep, each vertex first updates its score by summing up the scores of its neighbors which are pointed to by the vertex from the previous superstep, as shown in steps 8 to 12. Then each vertex updates its score

by summing up the scores of its neighbors pointing to the vertex from the previous superstep. We normalize the score array by its magnitude. Steps 25 and 26 update AC_{prev} using AC_{curr}, which will be used in the next superstep. The program stops if the *residual* is smaller than the *tolerance* or MAX_ITER is reached. We use the WebGraph framework to achieve random access to the compressed graph (steps 10 and 18) and Java 8 parallel stream to parallelize steps 8 to 12 and steps 16 to 20.

3.1.4 PageRank

PageRank [20] is an algorithm used by Google to measure the importance of web pages and to rank web pages in their search engine results. The algorithm considers two factors when measuring the importance of a web page: the number of links pointing the page and the quality of the links. If a page is linked to by many other pages with high PageRank scores, the page itself will receive a high rank. The mathematical definition is defined as follows.

PageRank (PR) of a vertex v_i in a graph $G = (V, E)$ is defined as

$$PR(v_i) = \frac{1-d}{n} + d \sum_{v_j \in IN(v_i)} \frac{PR(v_j)}{D(v_j)}, \tag{3.9}$$

where d is the damping factor (around 0.85), n is the total number of vertices, $IN(v_i)$ is the set of $v_i's$ in-neighbours (vertices with links to v_i), and $\mathrm{D}(v_j)$ is the out-degree for v_j.

In the matrix form, the PageRank vector

$$\vec{PR} = [PR(v_1), PR(v_2), \cdots, PR(v_n)] \tag{3.10}$$

can be viewed as the principal left eigenvector of the modified adjacency matrix. $\vec{PR}$ is the solution to the equation

$$\begin{aligned} \vec{PR} =& [\frac{1-d}{n}, \frac{1-d}{n}, \cdots, \frac{1-d}{n}] \\ &+ d \cdot \vec{PR} \begin{bmatrix} d(v_1, v_1) & d(v_1, v_2) & \cdots & d(v_1, v_n) \\ d(v_2, v_1) & \ddots & & \vdots \\ \vdots & & d(v_i, v_j) & \\ d(v_n, v_1) & \cdots & & d(v_n, v_n) \end{bmatrix} \end{aligned} \tag{3.11}$$

Algorithm 4 PageRank Compute Function

```
1:  function COMPUTE(G)
2:      if superstep = 0 then
3:          for v ∈ V do
4:              PR_prev[v] ← 1/n
5:      while superstep < MAX_ITER and residual > tolerance do
6:          superstep ← superstep + 1
7:          for v ∈ V do
8:              sum ← 0
9:              for u ∈ IN(v) do
10:                 sum ← sum + PR_prev[u]/out_degree[u]
11:             PR_curr[v] ← (1−d)/n + d · sum
12:         residual ← ‖P⃗R_curr − P⃗R_prev‖
13:         for v ∈ V do
14:             PR_prev[v] ← PR_curr[v]
```

where $d(v_i, v_j)$ is 0 if v_i does not link to v_j, and each row is normalized such that for each i

$$\sum_{j=1}^{N} d(v_i, v_j) = 1. \tag{3.12}$$

Equation 3.9 shows the intrinsic vertex-centric feature of the PageRank algorithm. The score of a vertex is only influenced by its neighbors close to it. We present the pseudocode for PageRank computation in Alg. 4. All vertices are initialized with the value of $1/n$ at superstep 0. n is the total number of vertices in the graph. The initial value will not affect the final score distribution. We assign the same score to each node with the assumption that at first each node does not know any information about other nodes. As the iteration continues, each node will gradually collect information from other nodes and the score distribution will gradually stabilize. For the subsequent supersteps, each vertex will sum all the messages (score divided by out-degree) received from its neighbours and update its value by Eq. 3.9. We maintain two arrays to record the vertex values for the previous and current supersteps in the program. We first update the current vertex score as shown in Step 10. Then Step 13 calculates the Euclidean distance between the two arrays as the residual. Lastly, Steps 14 and 15 update the previous vertex score using the current value, which will be read for the next superstep. The program stops if the residual is below the predefined tolerance or MAX_ITER is reached.

3.1.5 Betweenness Centrality

Betweenness centrality [21] is a measure of a vertex's centrality based on shortest paths. The measure quantifies, for each vertex, the number of shortest paths passing through the vertex; in other words, the number of times the vertex acting as a bridge along the shortest path between two other vertices. The betweenness centrality score represent the degree to which vertices stand between each other. For example, vertices with high betweenness scores are more central in the network and would have more control over the network, since more information will pass through those vertices. The formal definition of betweenness centrality is defined as follows.

Betweenness Centrality (BC) of a vertex v in a graph $G = (V, E)$ is

$$BC(v) = \sum_{s,t \in V, s \neq t \neq v} \delta_{st}(v), \tag{3.13}$$

where $\delta_{st}(v)$ is called **pair-dependency** of vertex v given a pair (s, t), and is defined as

$$\delta_{st}(v) = \frac{\sigma_{st}(v)}{\sigma_{st}}, \tag{3.14}$$

in which $\sigma_{st}(v)$ denotes the total number of the shortest paths from s to t that pass through v, and σ_{st} denotes the total number of the shortest paths from s to t.

3.1.5.1 Exact Computation

Brandes' algorithm [22] is currently the fastest algorithm for exact BC computation, which bases on the dependency of a source vertex on a given vertex.

Given a graph $G = (V, E)$, the **dependency** of a source vertex $s \in V$ on a vertex $v \in V$ is

$$\delta_s(v) = \sum_{t \in V, s \neq t \neq v} \delta_{st}(v). \tag{3.15}$$

Based on the above equation, the BC value of vertex v can be rewritten as

$$BC(v) = \sum_{s \neq v} \delta_s(v). \tag{3.16}$$

Brandes presented a recursive way to calculate the BC value of v, by introducing *predecessors* of v.

Given a graph (V, E), the **predecessors** of a vertex $v \in V$ on a shortest path

from s to v is a subset $P_s(v) \subseteq V$ s.t.

$$t \in P_s(v) \Rightarrow \big(d(s,v) = d(s,t) + 1\big) \wedge (t,v) \in E, \tag{3.17}$$

where $d(s,t)$ denotes the length of a shortest path from s to t.

Brandes' algorithm is based on the following theorem:

Given a graph (V,E), for any $s, v \in V$, we have

$$\delta_s(v) = \sum_{t \in V s.t. v \in P_s(t)} \frac{\sigma_{sv}}{\sigma_{st}}(1 + \delta_s(t)). \tag{3.18}$$

The basic idea of Brandes' algorithm can be summerized as follows:

1. For each vertex $s \in V$, we calculate the shortest paths from s to all the other vertices, i.e., using breadth-first search for unweighted graphs, the running time is bounded by $O(|E| + |V|)$; or using Dijkstra's algorithm for weighted graphs, which takes at least $O(|E| + |V| \log |V|)$ time.

2. For each vertex $s \in V$, traverse the vertices in descending order of their distances from s, and accumulate the dependencies by Eq. 3.18. Each traversal takes $O(|E|)$ time.

The total time complexity $T(n,m)$ for Brandes' algorithm, where $n = |V|$ and $m = |E|$, is therefore

$$T(n,m) = O(n(m+n) + nm) = O(nm) \tag{3.19}$$

for unweighted graphs, and

$$T(n,m) = O(n(m + n \log n) + nm) = O(nm + n^2 \log n) \tag{3.20}$$

for weighted graphs.

Equation 3.16 shows that betweenness score can be obtained by summing up the dependency values. The computation can be broken down to two stages: single-source shortest-path (SSSP) computation to count the number of shortest paths from the source to other vertices and the accumulation computation to obtain the dependency values. We present the pseudocode for betweenness computation in Alg 5. Supersteps are delimited by the minimum step distance from the source vertex. For example,

Algorithm 5 Betweenness Compute Function

```
1: function COMPUTE(G)
2:     for v ∈ V do BC[v] = 0
3:     for s ∈ V do
4:         /* single-source shortest-path */
5:         if depth = 0 then
6:             dist[s] ← 0; σ[s] ← 1
7:             for t ∈ V, t ≠ s do
8:                 dist[t] ← −1; σ[t] ← 0; δ[t] ← 0
9:         while does not reach the farthest vertex do
10:            for v at depth away from s do
11:                for w ∈ ON(v) do
12:                    /* path discovery*/
13:                    /* w visited for the first time */
14:                    if dist[w] = −1 then
15:                        dist[w] ← depth + 1
16:                    /* path counting*/
17:                    /* edget(v, w) on a shortest path */
18:                    if dist[w] = depth + 1 then
19:                        σ[w] ← σ[w] + σ[v]
20:            depth ← depth + 1
21:        /* accumulation */
22:        while depth > 0 do
23:            depth ← depth − 1
24:            for w at current depth do
25:                for v ∈ IN[w] do
26:                    /* v is a predecessor of w */
27:                    if dist[v] = depth − 1 then
28:                        δ[v] ← δ[v] + σ[v]/σ[w] · (1 + δ[w])
29:                if w ≠ s then BC[w] ← BC[w] + δ[w]
30:    /* rescaling */
31:    scale ← 1/((n − 1) · (n − 2))
32:    for v ∈ V do BC[v] ← BC[v] · scale
```

superstep 2 means we are processing vertices that are 2 steps away from the source vertex. The total number of superstpes is limited by the longest shortest path of the graph. Betweenness is initialized to 0. We use a *dist* array to record the distance between the source and other vertices and a σ array to record the number of shortest paths from the source vertex to the target vertex. The SSSP process (steps 5 – 20)

starts from the source vertex and traverses the graph layer by layer until reaching the farthest vertices. Along the way, we count the number of shortest paths from the source to a certain vertex. The accumulation process (steps 22 – 29) starts from the farthest vertices and traverses vertices in the descending order of their distances from the source, and accumulates the dependency values along the way. For each source vertex, the computation will contribute a summand to betweenness array. The final betweenness array will be obtained after executing such computation (SSSP with accumulation) on all vertices. We perform SSSP and accumulation processes for each vertex, which are independent among different vertices. We use parallel stream in Java 8 to process individual vertices (step 3) in parallel.

Although Brandes' algorithm is the fastest algorithm on computing the exact BC values, the time complexity can be extremely high for large graphs. This motivates us to investigate the BC approximate computation by implementing two approximation algorithms: uniformly random sampling [23] and adaptive sampling [24].

3.1.5.2 Approximate Computation

Uniformly Random Sampling The exact computation consists of solving n single-source shortest-paths (SSSP) problems, one for each vertex, and each SSSP contributes one summand to the result. This contribution is the one-sided dependency of the source $\delta_s(v)$ for betweenness. The vertices for which an SSSP is solved are called pivots. The basic idea for approximate computation is that the exact centrality value can be estimated by extrapolating the contributions obtained from just a few SSSP computations, i.e. from a small set of pivots.

If pivots are selected uniformly at random, the contributions of different SSSP computations to the BC value of a single vertex can be considered the result of a random experiment X_i. Therefore, the error bound for the estimated BC value of a single vertex (the average of the contributions from the randomly selected pivots) can be obtained by Hoeffding's inequality:

$$\Pr[\left|\frac{(X_1+\cdots+X_k)}{k} - E\left(\frac{X_1+\cdots+X_k}{k}\right)\right| \geq \xi] \leq e^{-2k(\frac{\xi}{M})^2}, \tag{3.21}$$

with $0 \leq X_i \leq M (i = 1, \cdots, k)$ and an arbitrary $\xi \geq 0$. Setting

$$M = n(n-2), \tag{3.22}$$

Algorithm 6 Betweenness (Uniformly Random Sampling)

```
1: function COMPUTE(G)
2:     P ← sample k vertices as pivots uniformly at random
3:     for s ∈ P do
4:         single-source shortest-path (same as Alg. 5)
5:         accumulation (same as Alg. 5)
6:     /* rescaling */
7:     scale ← n/((n − 1) · (n − 2) · k)
8:     for v ∈ V do BC[v] ← BC[v] · scale
```

Algorithm 7 Betweenness (Adaptive Sampling)

```
1: function COMPUTE(G)
2:     k ← 0
3:     while k < cuttoff do
4:         s ← sampling a vertex as the pivot
5:         single-source shortest-path (same as Alg. 5)
6:         accumulation (same as Alg. 5)
7:         k ← k + 1
8:         count ← number of vertices with betweenness larger than cn
9:         if count > topK then End While
10:    /* rescaling */
11:    scale ← n/((n − 1) · (n − 2) · k)
12:    for v ∈ V do BC[v] ← BC[v] · scale
```

$$\xi = \epsilon(n-2), \tag{3.23}$$

we can obtain an error bound from about ξ with probability of $2e^{-k(\frac{\epsilon}{n})^2}$. The pseudocode for uniform random sampling is shown in Alg. 6.

Adaptive Sampling Instead of setting the number of pivots k as one of the input parameters, the adaptive sampling technique determines the actual number of sampling k through each sampling. There is no need to predefine $k's$ value.

The basic idea is to repeatedly sample a vertex $v_i \in V$, perform SSSP from v_i, and maintain a running sum S of the dependency scores $\delta_{v_i}(v)$. Sample until S is greater than cn for some constant $c \geq 2$. Let the total number of samples to be k. The estimated centrality score of $v, BC(v)$ is given by $\frac{nS}{k}$.

In the practical implementation, we only focus on the vertices v with the high centrality scores ($BC(v) \geq cn$). Therefore, we need to specify the number of the top-score vertices $topK$ as one input parameter. We use $cuttoff$ to specify the maximum

number of samples. The pseudocode for adaptive sampling is shown in Alg. 7.

3.2 Diameter

Diameter and effective diameter computation is the basic problem for graph analytics. Diameter is defined as the longest shortest path length in the graph. Effective diameter is defined as the minimum number of steps in which 90% of all connected pairs of nodes can reach each other. Diameter and effective diameter are important properties of a graph for studying the interesting phenomena such as the famous "six degrees of separation" problem [25].

3.2.1 Exact Computation

We implement the exact computation of diameter based on its definition: the longest shortest path length. Shown in Alg. 8, for each node, we can perform a single-source shortest-path calculation using BFS starting this node. This is exactly the same process used in the exact betweenness computation algorithm. Therefore, we do not repeat the narration of the algorithm here. At the end of BFS, we can obtain the longest shorted path length from this node. This quantity is defined as the radius. Then we need to perform BFS on every node and find the maximum radius to get the diameter D. The BFS method only requires $O(nD)$ memory. But the time complexity is $O(nm)$. For large graphs, the time complexity is too high .

3.2.2 Approximate Computation

To reduce the time complexity, we can use dynamic programming. For example, for each node, we maintain a set to store its all reachable nodes within t steps. For $t+1$ step, the set for a node u can be obtained by taking the union of the sets of u's neighbors from the previous step t. The program runs iteratively ($t = 0$, $t = 1$,...) until the set for each node doesn't change, meaning we have arrived at the diameter. The method can reduce the time to $O(nD)$. But it requires to maintain an explicit set for each node. Thus, the space complexity is $O(n^2)$. For large graphs, the space complexity is too high.

The set is used to keep track of the number of nodes reachable within t steps as well as to do the union operation. For example, we can use a n-bit bit vector to

Algorithm 8 Diameter Compute Function

```
1: function COMPUTE(G)
2:     for s ∈ V do
3:         /* single-source shortest-path */
4:         if depth = 0 then
5:             dist[s] ← 0; σ[s] ← 1
6:             for t ∈ V, t ≠ s do
7:                 dist[t] ← −1; σ[t] ← 0
8:         while does not reach the farthest vertex do
9:             for v at depth away from s do
10:                for w ∈ ON(v) do
11:                    /* path discovery*/
12:                    /* w visited for the first time */
13:                    if dist[w] = −1 then
14:                        dist[w] ← depth + 1
15:                    /* path counting*/
16:                    /* edget(v, w) on a shortest path */
17:                    if dist[w] = depth + 1 then
18:                        σ[w] ← σ[w] + σ[v]
19:            depth ← depth + 1
20:        radius[s] = max{dist}
21:    diameter = max{radius}
```

encode the set. To reduce the space complexity, we can resort to the Flajolet-Martin probabilistic counters [26]. The probabilistic counters can estimate the cardinalities of the sets with *logn* bits.

3.2.2.1 Neighborhood Function

In this section, we define the neighborhood function, the cumulative distribution function of distances (distance cdf), and the effective diameter for a given graph $G = (V, E)$ [27].

(*ball of radius r*). The ball of radius r centered at vertex u is the set

$$B(u, 0) = \{u\} \tag{3.24}$$

$$B(u, r) = \bigcup_{(u,v)\in E} B(v, r-1) \tag{3.25}$$

that contains all vertices reachable from u within r steps.

The neighborhood function is defined based on the ball set:

(*neighborhood function*). The neighborhood function $N_G(t)$ is the number of node-pairs that can reach each other in at most t steps:

$$N_G(t) = \sum_{v \in V} |B(v,t)| \tag{3.26}$$

The cumulative distribution of distances (distance cdf) is defined as follows:

(*distance cdf*). The cumulative distribution function of distances $H_G(t)$ is the fraction of reachable node-pairs at distance t:

$$H_G(t) = \frac{N_G(t)}{N_G(t_{max})} \tag{3.27}$$

The diameter D of the graph (the longest shortest path) is the minimum t such that $H_G(t) = 1$.

(*effective diameter*). The effective diameter D_{eff} of a graph is defined as the minimum number of steps in which 90% of all connected pairs of nodes can reach each other:

$$D_{eff} = min\{t\} \; such \; that \; H_G(t) \geq 0.9 \tag{3.28}$$

3.2.2.2 Flajolet-Martin Counters

To approximate the diameter, we can use the Flajolet-Martin (FM) counters to count the number of distinct elements in a multiset since FM counters can give an unbiased estimate of the cardinality of a multiset and an $O(logn)$ bound for the space complexity. The following briefly describes how FM counters work.

We can assume that there is a mapping function that maps the elements of the set V into the set of bit strings of length L

$$mapping : V \rightarrow \{0,1\}^L \tag{3.29}$$

We can observe that if the output of the mapping function is uniformly distributed, the probability of getting a bit string with the pattern 0^k1 is 2^{-k-1}.

We define function $bit(b,k)$ return the k-th bit in the bit string b. We also define

a function $\rho(b)$ returns the position of the least significant 1-bit in the bit string b:

$$\rho(b) = \min_{k \geq 0} \{bit(b, k) = 1\} \tag{3.30}$$

We can use a bit vector $BITMAP$ of length L as the FM counter to keep track of the occurrences of such patterns. The procedure of adding an element u of the set V to the FM counter goes like this:

- Use the mapping function to obtain the bit string $map(u)$.
- Obtain the position of least significant 1-bit $i = \rho(map(u))$.
- Set the i-th bit of $BITMAP$ to 1: $BITMAP[i] = 1$.

The basic idea is that $BITMAP[0]$ will be accessed approximately $n/2$ times, $BITMAP[1]$ will be accessed approximately $n/4$ times and so on. At the end of execution, we could expect that $BITMAP[i]$ will be 0 if $i >> logn$ and 1 if $i << logn$. $i \approx logn$ will be the border of 0 and 1. If we let R be the position of the leftmost 0, we can use

$$n = \frac{1}{\phi} 2^R \tag{3.31}$$

to be the estimate of n, where $\phi \approx 0.77351$. If we use K $BITMAP$s and stochastic averaging, the estimate of n will be:

$$A = \frac{R_1 + R_2 + \cdots + R_K}{K} \tag{3.32}$$

$$n = \frac{1}{\phi} 2^A \tag{3.33}$$

We can use bias and standard error to evaluate the quality of the estimation. The bias is the ratio between the estimate of n and its exact value. The standard error is the quotient of the standard deviation of the estimate of n by the value n.

The quality of the FM counters is [26]:

$$bias = 1 + \frac{0.31}{K} \tag{3.34}$$

$$standard\ error = \frac{\sigma}{n} = \frac{0.78}{\sqrt{K}} \tag{3.35}$$

Shown in Alg. 9, we maintain K FM bitstrings $b(t, i)$ for each node i and current iteration number t. $b(t, i)$ encodes the number of nodes reachable from node i within t

Algorithm 9 Diameter Approximating Function

```
1: function COMPUTE(G)
2:     for i = 1 to n do
3:         b(0, i) ← NewFMBitstring()
4:     for t = 1 to MaxIter do
5:         Changed ← 0
6:         for i = 1 to n do
7:             for l = 1 to K do
8:                 for j ∈ i's neighbor do
9:                 b_l(t, i) ← b_l(t − 1, i) BIT-OR b_l(t − 1, j)
10:                if b_l(t, i) ≠ b_l(t − 1, i) then Changed ← Changed + 1
11:        N_G(t) ← Σ_i N_G(t, i)
12:        if Changed = 0 then t_max ← t and break
13:    diameter ← t_max
14:    diameter_eff ← smallest t where N_G(t) ≥ N_G(t_max)
```

steps. The bitstrings $b(t, i)$ are iteratively updated until all bitstrings stabilize. Steps 6–9 show each node i updates its bitstring by performing bitwise OR on all bitstrings of its neighbors handed over from the previous iteration. The neighborhood function $N_G(t, i)$ for node i after t steps can be estimated by:

$$N_G(t, i) = \frac{1}{0.77351} 2^{\frac{1}{K} \sum_{l=1}^{K} b_l(i)} \tag{3.36}$$

where $b_l(i)$ is the position of leftmost '0' bit of the l^{th} bitstring of node i. The iteration stops when the bitstrings of all nodes stabilize (step 12). Then t_{max} is the diamter of the graph. We can calculate the effective diameter which is the smallest t such that $N_G(t) \geq 0.9 \cdot N_G(t_{max})$.

3.3 Truss Decomposition

Identifying various cohesive subgraphs in a massive network is crucial to the efficient and effective analytics of the network [28–31]. k-truss is an important kind of cohesive subgraphs of a network that has received growing attention in the recent years [32–36]. Motivated by the need to find a structure that is a relaxation of a clique [37] and is efficiently computable, k-truss finds applications in social network visual analysis [38], community search [39], maximum clique finding [40], etc. The k-truss of a graph is defined as the largest subgraph in which each edge is contained in at least $k - 2$

triangles within the subgraph [41]. Given a graph, the k-truss decomposition problems aims to find the k-trusses of the graph for all k.

The definition of k-truss is similar to k-core [11, 42–45], which is defined as the largest subgraph in which every vertex has a minimum degree of k within the subgraph. The k-truss focuses on the edges of a graph while the k-core focuses on the vertices. We can make an analogy that the edge in k-truss is similar to the vertex in k-core while the number of triangles for an edge to be contained in is similar to the degree for a vertex. However, the definition of k-truss is more rigorous than k-core since k-truss is based on triangles, which have higher dimensionality than edges that k-core is based on.

There are mainly two types of algorithms for efficiently computing the k-trusses: the serial algorithm suited for medium-sized graphs and the parallel algorithm suited for large-scale graphs. The serial algorithm is based on the concept of edge peeling proposed by J. Wang and J. Cheng [41]. Their algorithm iteratively eliminates edges at each stage based on their support value until all edges in the graph are removed. In their implementation, a hash table was used to check whether two vertices form an edge or not. The endpoints of each edge were hashed as the keys for the hash table. The hash table can work well for moderate-sized graphs. However, for large graphs, the hash table is expensive to use and designing an optimum hash function is not a trivial problem. The second type of algorithms use more advanced parallelization techniques on high-performance multi-core machines to significantly reduce the runtime [46–49]. Memory usage is not the major concern for these parallel programs since they are designed for high-performance machines, which are usually capable of keeping the whole graph as well as the hash table in the main memory. However, the cost for the hardware is high. For algorithms that avoid using the hash table (*e.g.*, [46] uses an array-based alternative), we can still find room for optimization on the data structure design to use the memory more efficiently. In short, both the serial and the parallel algorithms have limitations. For the serial algorithm, the inefficient and cumbersome data structure design (*e.g.*, the use of a hash table) hinders its use for large graphs. For the parallel algorithms, besides the same problem that the serial algorithm suffers, the other problem is the high hardware cost, as the focus of the parallel algorithms is to reduce the runtime on powerful machines.

Different from the algorithms proposed in the IEEE HPEC static graph challenge using high performance CPU or GPU to boost performance [50], our focus is to investigate if it is viable to efficiently and economically compute the k-trusses of

large networks on a single consumer-grade machine. Therefore, the memory usage by the program is our major concern. We aim to engineer two algorithms: the serial edge-peeling algorithm [41] and the parallel asynchronous h-index-updating algorithm [48], with the goal to minimize the memory usage compared to the original implementations, but still with high time efficiency. We target these two algorithms because of their efficiency and relatively small memory footprints. For example, the edge-peeling algorithm optimizes Cohen's very first k-truss decomposition algorithm [32] with improved time complexity. The asynchronous h-index-updating algorithm has a relatively smaller memory footprint compared to other parallel algorithms. A concrete example would be: for a graph with 41 million vertices and 1.2 billion edges, the h-index-updating algorithm needs around 24 GB memory while the memory-efficient parallel algorithm in [46] needs around 34 GB memory.

For the k-truss problem, or any graph-related problem, keeping the complete graph representation in the main memory is commonly the most memory-consuming component. To significantly reduce the graph's footprint in the main memory, we resort to WebGraph [8], a highly efficient graph compression framework that allows random access to a memory-mapped compressed graph stored on the hard drive. WebGraph also supports thread-safe operations on an immutable graph, facilitating parallel computation. The other memory-consuming component in the k-truss program is the use of a hash table to check whether two vertices form an edge or not. The hash table has a total number of entries equal to the number of edges of the graph. The purpose of using a hash table is to achieve constant-time querying in an optimum scenario. However, it is expensive to use for large graphs in practice. Therefore, in our implementation, we avoid using a hash table. We carefully design an array-based structure with a small memory footprint and its corresponding operations to achieve the same functionality that a hash table can provide. With our optimized implementation, we can efficiently compute the k-trusses of large networks (up to 1.2 billion edges) on a consumer-grade machine.

3.3.1 Preliminaries

For the k-truss decomposition problem, we consider undirected, unweighted simple graphs. For a given graph G, the vertex set is denoted by V and the edge set is denoted by E. Therefore, the number of vertices is $n = |V|$ and the number of edges is $m = |E|$. The set of neighbors of a vertex u is denoted by $neighbor(u)$ such

that $neighbor(u) = \{v : (u, v) \in E\}$. The degree of u is defined as $degree(u) = |neighbor(u)|$.

Each vertex in the graph is assigned an unique vertex ID from 0 to $n - 1$. The order of vertices is based on their vertex IDs. For example, we say u is ordered before v if $u < v$. Based on this ordering, we define a triangle as follows:

(*triangle*). A triangle in G is defined as a cycle of three vertices $\{u, v, w \in V\}$, denoted by $\triangle_{uvw}$, such that $u < v < w$ and all three edges exist in G (*i.e.*, $(u, v), (v, w), (u, w) \in E$).

With the notion of triangles, we introduce the definition for support of an edge:

(*support*). The support of an edge $e \in E$, denoted by $support(e)$, is defined as the number of triangles in G that contain e. k-truss is defined based on the notion of support: (*k-truss*). The k-truss of G, denoted by T_k, where $k \geq 2$, is defined as the largest subgraph of G, such that every edge e in T_k has $support(e) \geq (k - 2)$.

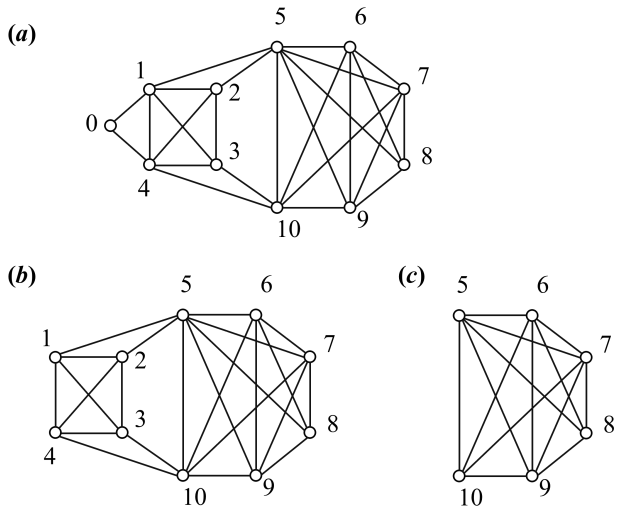

Figure 3.1: (a) An undirected, unweighted simple graph G; (b) the 4-core of G (no 5-core exists); (c) the 5-truss of G (no 6-truss exists).

k-truss has close connections with the well known concept of k-cores. The k-core of G, denoted by C_k, where $k \geq 0$, is defined as the largest subgraph of G, such that each vertex u in C_k has $degree(u) \geq k$. Fig. 3.1(a) shows a simple undirected and unweighted graph G. By definition, the 2-truss is simply G itself. Fig. 3.1(b) shows the 4-core of G in which every vertex has a degree of at least 4. No 5-core of G exists. Fig. 3.1(c) shows the 5-truss of G in which every edge has a support of at least 3. No 6-truss of G exists. It is interesting to note that the 5-truss also satisfies the requirement of a 4-core by definition. However, it is not true vice versa. This

example shows that the k-truss can further filter out those marginal vertices and can better represent the core part of a graph than the k-core.

We introduce other two important notions related to k-truss: trussness and k-class.

(*trussness*). The trussness of an edge e, denoted by $\phi(e)$, is defined as the maximum k such that e belongs to T_k but does not belong to T_{k+1}.

The maximum trussness of any edge in G is denoted by t_{max}. Based on the trussness, we can define the k-class of G as follows: (*k-class*). The k-class of G is defined as the set of edges with same trussness of k, denoted by $\Phi_k = \{e : e \in E, \phi(e) = k\}$.

For the k-truss decomposition problem, our task is to find the k-trusses of G for all $2 \leq k \leq t_{max}$. The k-truss can be obtained by taking the union of k-classes of G by $T_k = \Phi_k \cup \Phi_{k+1} \cup \cdots \cup \Phi_{t_{max}}$. For the graph shown in Fig. 3.1(a), the 2-class Φ_2 is an empty set. The 3-class Φ_3 has 6 edges: {(0, 1), (0, 4), (1, 5), (2, 5), (3, 10), (4, 10)}. The 4-class Φ_4 has 6 edges: {(1, 2), (1, 3), (1, 4), (2, 3), (2, 4), (3, 4)}. The 5-class Φ_5 has 14 edges: {(5, 6), (5, 7), (5, 8), (5, 9), (5, 10), (6, 7), (6, 8), (6, 9), (6, 10), (7, 8), (7, 9), (7, 10), (8, 9), (9, 10)}. Therefore, from the k-classes above, we can obtain that T_2 and T_3 is G itself. T_4 is $(\Phi_4 \cup \Phi_5)$ and T_5 is Φ_5. We can easily verify that for $2 \leq k \leq 5$, each edge e in T_k is contained in at least $(k-2)$ triangles, meaning that $support(e) \geq (k-2)$.

To summarize, if we can compute the trussness for each edge in G, we can obtain the k-classes of G, and then we can obtain the k-trusses of G by taking the union of the k-classes. Therefore, the k-truss decomposition of a graph is equivalent to computing the trussness of each edge in the graph.

3.3.2 Initial Support Computation

We engineer two existing efficient k-truss decomposition algorithms: the serial algorithm based on edge peeling [41] and the parallel algorithm based on asynchronous h-index updating [48]. Both algorithms start with computing the initial support for each edge since the initial support is the upper bound of the trussness. Since the memory usage is our major concern, we surely do not want to load the whole graph into the main memory during computation, especially for large networks. Therefore, we use WebGraph [8], a graph compression framework with high compression ratio that enables random access to the compressed graph, to make the graph's footprint

Algorithm 10 Initial Support Computation

```
1:  function supportCompute(G)
2:      for each edge e ∈ E do support[e] ← 0
        each edge e ∈ E
3:      u, v ← two endpoints of e
4:      for each w ∈ neighbor+(u) ∩ neighbor+(v) do
5:          e_uw ← get edge ID of (u, w)
6:          e_vw ← get edge ID of (v, w)
7:          atomicAdd(support[e], 1)
8:          atomicAdd(support[e_uw], 1)
9:          atomicAdd(support[e_vw], 1)
10:     return support
```

as small as possible. We also design an array-based structure and corresponding operations to achieve the equivalent functionality of a hash table, but with a much smaller footprint. We introduce our memory optimization on the two k-truss decomposition algorithms using WebGraph and our carefully engineered data structures in the following sections.

The initial support is the upper bound on the trussness of each edge. Its computation is based on triangle enumeration [46]. This algorithm visits each edge in the graph, finds all triangles starting with the edge, and updates the support for all edges contained in those triangles.

Since we define a triangle $\triangle_{uvw}$ based on the ordering of the three vertices ($u < v < w$), to find triangles starting with edge (u, v), we do not need the complete neighbor sets of u and v. We define the set of u's neighbors with vertex ID $> u$ to be the upper neighbor set of u, denoted by $neighbor^+(u) = \{v : (u, v) \in E, v > u\}$. To find all triangles starting with (u, v), we take the intersection of $neighbor^+(u)$ and $neighbor^+(v)$. uvw forms a triangle $\triangle_{uvw}$ if $w \in \{neighbor^+(u) \cap neighbor^+(v)\}$.

The initial support computation can be parallelized easily since the procedure of triangle enumeration is independent on different edges. Algorithm 10 summarizes the major steps of the initial support computation. For each edge $e \in E$, step 4 obtains the two endpoints u and v of the edge. Step 5 takes the intersection of u and $v's$ upper neighbor sets. The size of the intersection is the number of triangles starting with edge (u, v). Steps 6 and 7 obtain the edge IDs for (u, w) and (v, w) according to the endpoints. The original program [41] uses a hash table to store the edge IDs. The pair of two endpoints of an edge is used as the key for the hash table. The hash table is also used to check whether two vertices form an edge or not. Although it

is convenient to use, for large graphs, the hash table is expensive to use in practice. Instead, we implement these two steps using binary search in the adjacency list of the graph. We use an auxiliary array to label the starting positions of each segment in the adjacency list. Therefore, we can perform binary search in a small range (not in the whole adjacency list), which can be very efficient. Steps 8–9 atomically increment the support of the three edges constituting the triangle $\triangle_{uvw}$ by 1.

3.3.3 Serial Edge Peeling

Algorithm 11 Serial K-Truss Decomposition

```
1: function k-TRUSS-SERIAL(G, support)
2:     k ← 2
3:     sort all the edges in ascending order of their support [1] and store them in
       sortedEdge array
4:     for each edge e ∈ E such that support[e] ≤ k − 2 do
5:         e ← edge with the lowest support
6:         u, v ← two endpoints of e
7:         if degree[u] > degree[v] then swap u and v
8:         for each w ∈ neighbor(u) do
9:             e_uw ← get edge ID of (u, w)
10:            if (v, w) ∈ E then
11:                e_vw ← get edge ID of (v, w)
12:                if support[e_vw] > support[e] then
13:                    support[e_vw] ← support[e_vw] − 1
14:                    reorder sortedEdge
15:                if support[e_uw] > support[e] then
16:                    support[e_uw] ← support[e_uw] − 1
17:                    reorder sortedEdge
18:        remove e from G.
19:    if not all edges in G are removed then
20:        k ← k + 1
21:        goto step 4
22:    for edge e ∈ E do
23:        trussness[e] ← support[e] + 2
24:    return trussness
```

The serial edge-peeling algorithm uses the output from Algorithm 10 to initialize the support for each edge. Algorithm 11 iteratively removes edges based on their support until all edges in the graph are removed.

Step 3 sorts the edges in ascending order of their support using a linear-time sort (*e.g.*, bin sort) and stores them (edge IDs) in the *sortedEdge* array. Edges are processed exactly once under the ascending-order configuration. Step 5 obtains the edge e (not removed) with the lowest support from *sortedEdge*. Step 7 ensures that u has a smaller degree than v. The removal of edge (u, v) affects the support of all edges that can constitute triangles with (u, v). To find all triangles containing edge (u, v), step 8 iterates $u's$ each neighbor w. If (v, w) is an edge, then u, v, and w form a triangle. Step 10 checks whether (v, w) is an edge or not. We implement this step by binary search in the adjacency list of the graph without a hash table, similar to the operation of obtaining an edge ID. If (v, w) is an edge, then steps 13 and 16 decrement the support of edges (v, w) and (u, w) by 1, respectively. It should be noted that the decrement only applies to edges with support larger than edge $e's$ support. Since the support has been changed, steps 14 and 17 reorder the *sortedEdge* array to maintain the ascending order with regard to the edge support. Constant-time reordering can be achieved using a method similar to the one used in the k-core decomposition [42]. After all triangles containing edge (u, v) are processed, step 18 removes the edge (u, v) from the graph. We do not physically delete the edge from the graph. Instead, we use a bit set to label each edge's state. After removing all edges in the current bin (containing edges with support equal to $k - 2$), the program moves to the next bin (incrementing k by 1). The program continues removing edges until all edges in the graph are removed. Step 23 adds 2 to the final support to obtain the trussness for each edge.

3.3.4 Asynchronous h-index updating

The edge-peeling method requires processing edges in ascending order of their support, which makes the algorithm inherently sequential since each step depends on the result from the previous step. The asynchronous h-index updating algorithm [48] relaxes the "ascending order" requirement and processes edges in a random order, which makes the parallelization possible. The main idea of the algorithm is the iterative h-index computation on the support of the edges. h-index is a measure to quantify the impact and productivity of researchers by the number of citations of their publications. For a set of real numbers, the h-index of the set is defined as the largest number h such that there are at least h elements in the set that are at least h. For example, the h-index of {2, 2, 3, 3, 3} is 3. The algorithm extends the definition

Algorithm 12 Parallel K-Truss Decomposition

```
1: function k-TRUSS-PARALLEL(G, support)
2:     for each edge e ∈ E do
3:         h[e] ← support[e], scheduled[e] ← TRUE
4:     updated ← TRUE                          ▷ TRUE if any h[e] is updated
5:     while updated do
6:         update ← FALSE each edge e ∈ E
7:         if scheduled[e] is FALSE then continue
8:         L ← empty list, N ← empty list
9:         for each △ contains e do
10:            e', e'' ← the two edges in △ other than e
11:            N.add(e'), N.add(e'')
12:            ρ ← min{h[e'], h[e'']}
13:            L.add(ρ)
14:        H ← h-index of L
15:        if h[e] ≠ H then
16:            updated ← TRUE
17:            for each edge e_N in N do
18:                if H < h[e_N] ≤ h[e] then
19:                    scheduled[e_N] ← TRUE
20:        h[e] ← H
21:        scheduled[e] ← FALSE
22:    for each edge e ∈ E do
23:        trussness[e] ← h[e] + 2
24:    return trussness
```

of neighbors and defines an edge's neighbors to be the edges that can form triangles with it. The h-index of an edge is fundamentally the same as the support of an edge and is upper bounded by the h-index of the edge's neighbor set. The algorithm iteratively updates an edge's h-index by computing the h-index of its neighbor set until achieving convergence when no updates would happen. The updating scheme is asynchronous, meaning the h-index of an edge is updated instantly and the computation of the h-index of the neighbor set always uses the up-to-date h-index values.

The major steps of the parallel algorithm are summarized in Algorithm 12. Step 3 initializes the h-index of each edge by the edge's initial support. We use a boolean indicator *updated* to check the convergence and to terminate the program. *updated* stays true if any edge's h-index is changed. The program processes each edge in parallel. For each edge e, step 10 finds all triangles containing the edge e. We use the

same implementation as steps 8 and 10 of Algorithm 11, which uses binary search in the adjacency list of the graph without a hash table. Steps 12–14 collects the h-indices of e's neighbor edges by choosing the smaller one. Step 15 computes the h-index of the neighbor set as the new h-index for edge e. If the new h-index is smaller than $e's$ current h-index, it means $e's$ h-index will be updated and step 17 sets *updated* to be true. The algorithm uses a notification mechanism to achieve faster convergence. The *scheduled* array notifies the program to process the scheduled edges only. Steps 18–20 decide whether a neighbor edge of e is scheduled to be processed in the next iteration. If the neighbor edge's h-index is between e's new h-index and $e's$ current h-index, it means that updating $e's$ h-index will affect the upper bound of the neighbor edge's h-index. The neighbor edge's h-index may have a chance to change. Therefore, it is scheduled to be processed in the next iteration. The program terminates when each edge's h-index achieves convergence. Step 24 adds 2 to the final h-index to obtain the trussness for each edge.

3.3.5 Memory Optimization

The original implementations for Algorithms 11 and 12 keep the entire graph (*i.e.*, the adjacency list representation) in the main memory and use a hash table for checking the existence of an edge and querying for the edge ID, which is expensive and inefficient for large graphs. To eliminate the necessity of keeping the entire graph in the main memory, we use the WebGraph framework. In this section, we detail our data structure design to avoid using the hash table but still can achieve the equivalent functionality without any performance degradation.

Fig. 3.2 shows our optimized data structures for k-truss decomposition. We use a small simple graph as an example. For the example graph, $n = 6$ and $m = 9$. The edges of the graph can be stored in $edgeHead$ and $edgeTail$ arrays both with size m. Vertices in $edgeHead$ and $edgeTail$ are sorted in ascending order. $edgeID$ in the original programs is a hash table with m entries. We show that we can remove the redundant $edgeID$ and $edgeHead$ without affecting any computation.

We introduce an *offset* array with size n in our implementation. The *offset* array summarizes the $edgeHead$ information by recording the start position of each vertex in $edgeHead$. For example, vertex 0 starts at position 0, vertex 1 starts at position 1, and vertex 2 starts at position 3 in $edgeHead$. Therefore, $offset[0] = 0$, $offset[1] = 1$, and $offset[2] = 3$, respectively. The hash table has two functionalities: given two

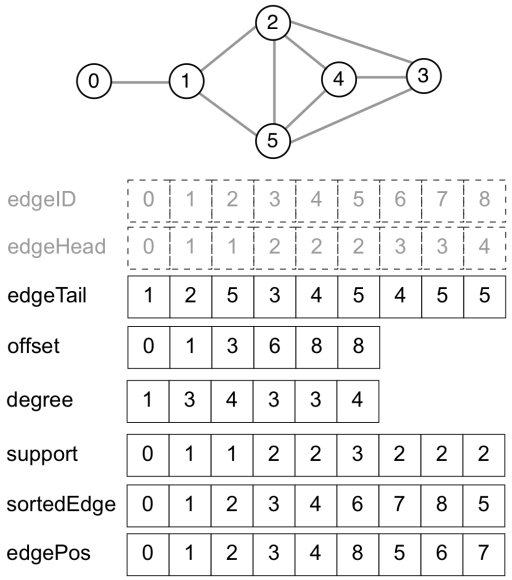

Figure 3.2: Optimized data structures for k-truss decomposition.

vertices u and v, check if (u, v) is an edge; given an edge (u, v), get the edge ID. We show that only by $edgeTail$ and *offset*, we can still achieve the two functionalities. For checking if (u, v) is an edge, we assert that $u < v$. Then we perform binary search in $edgeTail$ for v in the range of $[offset[u], offset[u+1])$, which is $neighbor^+(u)$. If we can find v in this range, then (u, v) is an edge. For example, if we want to check if (2, 5) is an edge, we can search for 5 in range [3, 6) in $edgeTail$. The other functionality is to get the edge ID given the two endpoints. The process is similar since the binary search would return the target index.

$sortedEdge$ is used to store edges in ascending order of their support. $edgePos$ is used to store the index of an edge in $sortedEdge$. For example, $edgePos[5] = 8$, meaning that edge 5 is at position 8 in $sortedEdge$ ($sortedEdge[8] = 5$). For Step 10 in Algorithm 11 and Step 11 in Algorithm 12, we still need the full neighbor set of a vertex. Since we cannot get the complete neighbor set from $edgeTail$, we therefore use WebGraph for random access to the compressed graph to retrieve the complete neighbor set of a vertex. Since Algorithm 12 processes edges in a random order, there is no need to keep $sortedEdge$ and $edgePos$. However, the *scheduled* needs another m-sized array.

By eliminating $edgeID$ and $edgeHead$, we optimize the memory usage down to $(4m+2n)$ for Algorithm 11 and $(3m+2n)$ for Algorithm 12, compared to the original programs' $(6m+n)$ and $(5m+n)$ memory usage (no *offset* needed). Since m is usually

very large, the memory reduction would be significant for large graphs. For example, for a graph with 41 million vertices and 1.2 billion edges, the original programs for Algorithm 2 and 3 would consume at least 29.0 GB and 24.1 GB memory, if we assume the unit is an integer. Our optimized programs only need 19.5 GB and 14.7 GB memory, respectively. It should be noted that an m-entry hash table usually has a larger memory footprint than an m-length array. Therefore, in practice, the memory reduction should be larger than the theoretical calculation.

WebGraph plays an important role in reducing the memory usage. Without WebGraph, we have to keep the complete graph (*i.e.*, the adjacency list representation) in the memory to retrieve the neighbor set of a vertex. By the WebGraph's API, we can efficiently get access to the compressed graph stored on the hard drive. Therefore, WebGraph eliminates the necessity of maintaining the complete graph in the main memory, allowing us to remove the $edgeHead$ array.

3.4 Summary

This chapter details the implementation of graph algorithms computing centrality measures (eigenvector, hub, authority, PageRank and betweenness), diameter, and truss-decomposition in Graph-XLL. For betweenness and diameter, besides the exact computation, we also implement the approximate calculation with reduced time and space complexity, while neither igraph nor NetworkX provides such approximate algorithms. For truss decomposition, we implement both sequential and parallel algorithms while neither igraph nor NetworkX provides truss decomposition programs. In next chapter, we will present the performance results using Graph-XLL on different datasets.

Chapter 4

Experiments

This chapter presents the experimental results using algorithms in Graph-XLL to compute centrality measures (eigenvector, hub, authority, PageRank and betweenness), diameter, truss decomposition for different datasets. Experiments are conducted on a machine with Intel quad-core i7, 2.6 GHz CPU, 32 GB RAM, and 500 GB SSD hard drive, running Ubuntu 17.10. The cost for this machine is less than 1,500 Canadian dollars, thus qualifying as a consumer-grade machine. We perform experiments on different datasets obtained from http://law.di.unimi.it/datasets.php. Characteristics of the datasets are summarized in Table 4.1.

Table 4.1: Summary of Datasets

Graph	$\|V\|$	$\|E\|$
p2p	62,586	147,892
slashDot	82,168	948,464
cnr-2000	325,557	3,216,152
eu-2005	862,664	19,235,140
dblp-2011	986,324	3,353,618
in-2004	1,382,908	16,917,053
ljournal-2008	5,363,260	79,023,142
eu-2015-host	11,264,052	386,915,963
arabic-2005	22,744,080	639,999,458
gsh-2015-tpd	30,809,122	602,119,716
uk-2005	39,459,925	783,027,125
twitter-2010	41,652,230	1,468,365,182

4.1 Centrality Measures

We use 7 datasets of varing sizes (cnr-2000, eu-2005, in-2004, ljournal-2008, eu-2015-host, arabic-2005, and twitter-2010) to compute the centrality measures. We record runtime and memory consumption when executing the programs. We present experimental results in this chapter. Further evaluation and analysis will be done in the next chapter.

4.1.1 Eigenvector Centrality

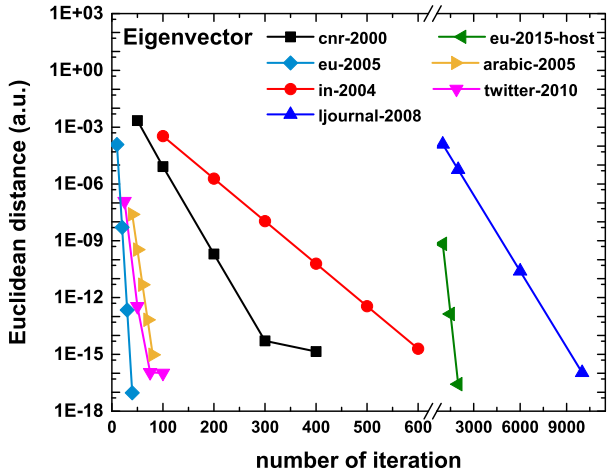

Figure 4.1: Euclidean distance of eigenvector centrality between two consecutive iterations after a certain number of iterations for different datasets. The Euclidean distance can be viewed as the convergence condition. If we choose 1E-14 as the criterion for convergence, the numbers of iterations required to achieve convergence are ∼300 for cnr-2000, ∼35 for eu-2005, ∼550 for in-2004, ∼9000 for ljournal-2008, ∼1500 for eu-2015-host, ∼70 for arabic-2005, and ∼60 for twitter-2010, respectively.

Fig. 4.1 shows the Euclidean distance of eigenvector centrality between two consecutive iterations after a certain number of iterations for different datasets. The Euclidean distance can be viewed as the convergence condition. If we choose 1E-14 as the criterion for convergence, the numbers of iterations required to achieve convergence are ∼300 for cnr-2000, ∼35 for eu-2005, ∼550 for in-2004, ∼9000 for ljournal-2008, ∼1500 for eu-2015-host, ∼70 for arabic-2005, and ∼60 for twitter-2010, respectively. To our surprise, the largest dataset twitter-2010 only requires a small number of iterations to achieve convergence. By contrast, the medium-sized datasets ljournal-2008 and eu-2015-host converge very slowly. The runtime to achieve convergence and

memory consumption for different datasets are summarized in Table 4.2.

Table 4.2: Runtime and Memory Consumption for Eigenvector

Graph	Iteration	Runtime (s)	Memory (GB)
cnr-2000	300	27	0.39
eu-2005	35	19	0.51
in-2004	550	215	0.67
ljournal-2008	9000	18,787	0.78
eu-2015-host	1500	8,556	1.0
arabic-2005	70	796	2.02
twitter-2010	60	3247	3.78

4.1.2 Hub Centrality

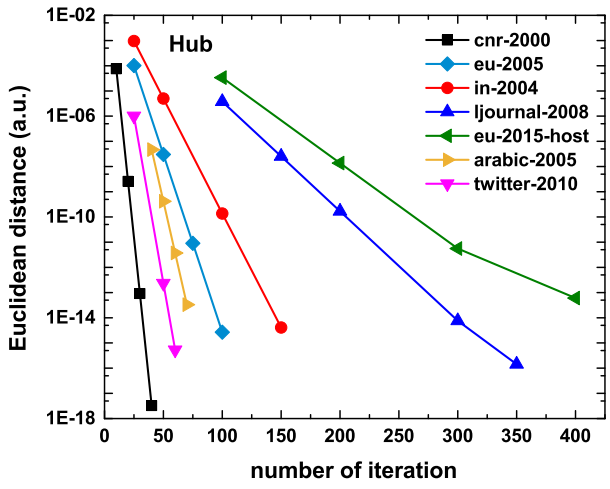

Figure 4.2: Euclidean distance of hub centrality between two consecutive iterations after a certain number of iterations for different datasets. The Euclidean distance can be viewed as the convergence condition. If we choose 1E-14 as the criterion for convergence, the numbers of iterations required to achieve convergence are ∼30 for cnr-2000, ∼90 for eu-2005, ∼140 for in-2004, ∼300 for ljournal-2008, ∼400 for eu-2015-host, ∼70 for arabic-2005, and ∼60 for twitter-2010, respectively.

Fig. 4.2 shows the Euclidean distance of hub centrality between two consecutive iterations after a certain number of iterations for different datasets. The Euclidean distance can be viewed as the convergence condition. If we choose 1E-14 as the criterion for convergence, the numbers of iterations required to achieve convergence are ∼30 for cnr-2000, ∼90 for eu-2005, ∼140 for in-2004, ∼300 for ljournal-2008, ∼400

for eu-2015-host, ∼70 for arabic-2005, and ∼60 for twitter-2010, respectively. The medium-sized graphs ljournal-2008 and eu-2015-host still experience the slow convergence issue while the largest dataset twitter-2010 can achieve convergence within a small number of iterations. The runtime to achieve convergence and memory consumption for different datasets are summarized in Table 4.3.

Table 4.3: Runtime and Memory Consumption for Hub

Graph	Iteration	Runtime (s)	Memory (GB)
cnr-2000	30	9	0.37
eu-2005	90	72	0.52
in-2004	140	117	0.56
ljournal-2008	300	1,160	1.0
eu-2015-host	400	4,226	1.1
arabic-2005	70	1,416	2.72
twitter-2010	60	5,785	3.41

4.1.3 Authority Centrality

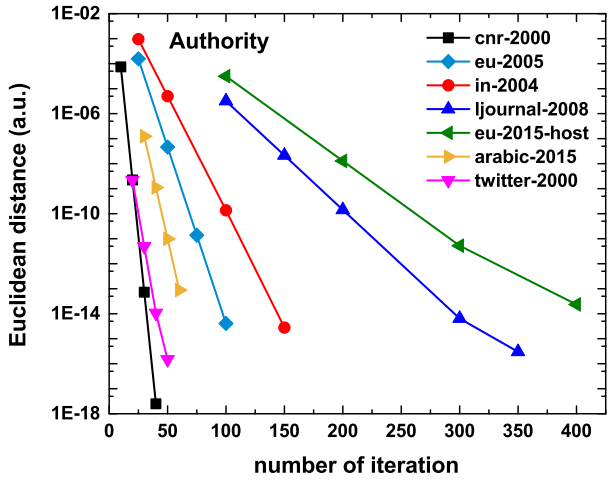

Figure 4.3: Euclidean distance of authority centrality between two consecutive iterations after a certain number of iterations for different datasets. The Euclidean distance can be viewed as the convergence condition. If we choose 1E-14 as the criterion for convergence, the numbers of iterations required to achieve convergence are ∼30 for cnr-2000, ∼90 for eu-2005, ∼140 for in-2004, ∼300 for ljournal-2008, ∼400 for eu-2015-host, ∼65 for arabic-2005, and ∼40 for twitter-2010, respectively.

Fig. 4.3 shows the Euclidean distance of authority centrality between two consecutive iterations after a certain number of iterations for different datasets. The

Euclidean distance can be viewed as the convergence condition. If we choose 1E-14 as the criterion for convergence, the numbers of iterations required to achieve convergence are ∼30 for cnr-2000, ∼90 for eu-2005, ∼140 for in-2004, ∼300 for ljournal-2008, ∼400 for eu-2015-host, ∼65 for arabic-2005, and ∼40 for twitter-2010, respectively. The medium-sized graphs ljournal-2008 and eu-2015-host still experience the slow convergence issue while the largest dataset twitter-2010 can achieve convergence within a small number of iterations. The runtime to achieve convergence and memory consumption for different datasets are summarized in Table 4.4.

Table 4.4: Runtime and Memory Consumption for Authority

Graph	Iteration	Runtime (s)	Memory (GB)
`cnr-2000`	30	9	0.4
`eu-2005`	90	73	0.51
`in-2004`	140	121	0.67
`ljournal-2008`	300	1,140	0.66
`eu-2015-host`	400	4,204	1.18
`arabic-2005`	65	1,356	2.74
`twitter-2010`	40	3,980	3.56

4.1.4 PageRank

Fig. 4.4 shows the Euclidean distance of PageRank between two consecutive iterations after a certain number of iterations for different datasets. The Euclidean distance can be used as the convergence condition. If we choose 1E-14 as the criterion for convergence, the numbers of iterations required to achieve convergence are ∼150 for cnr-2000, in-2004 and arabic-2005, ∼140 for eu-2005 and eu-2015-host, ∼135 for ljournal-2008, and ∼130 for twitter-2010, respectively. It is interesting to note that different datasets converge at the similar rate for PageRank. The runtime to achieve convergence and memory consumption for different datasets are summarized in Table 4.5.

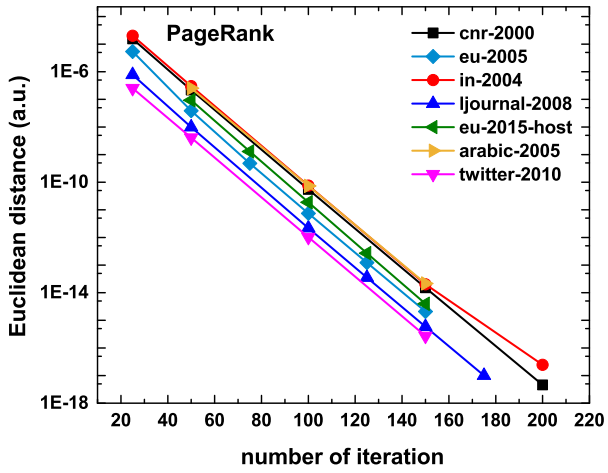

Figure 4.4: Euclidean distance of PageRank centrality between two consecutive iterations after a certain number of iterations for different datasets. The Euclidean distance can be viewed as the convergence condition. If we choose 1E-14 as the criterion for convergence, the numbers of iterations required to achieve convergence are ∼150 for cnr-2000, in-2004 and arabic-2005, ∼140 for eu-2005 and eu-2015-host, ∼135 for ljournal-2008, and ∼130 for twitter-2010, respectively.

Table 4.5: Runtime and Memory Consumption for PageRank

Graph	Iteration	Runtime (s)	Memory (GB)
`cnr-2000`	150	25	0.5
`eu-2005`	140	92	0.57
`in-2004`	150	111	0.65
`ljournal-2008`	135	677	0.7
`eu-2015-host`	140	2,618	1.11
`arabic-2005`	150	2,846	2.42
`twitter-2010`	130	18,883	3.71

4.1.5 Betweenness

4.1.5.1 Exact Computation

Due to the high time complexity for large graphs, we compute the exact betweenness centrality for small and medium-sized graphs (cnr-2000, eu-2005 and in-2004). Table 4.6 summarizes the runtime and memory consumption for exact betweenness computation. The exact betweenness centrality is compute-intensive. For example, using Graph-XLL, the runtime of computing the exact betweenness takes 5 h for cnr-2000. To address the time-consuming issue, in Graph-XLL, we provide two approximation algorithms (uniformly random sampling and adaptive sampling), which can reduce

the runtime significantly. Neither igraph nor NetworkX provides such approximation algorithms.

Table 4.6: Runtime and Memory Consumption for Exact Betweenness

Graph	Runtime (h)	Memory (GB)
cnr-2000	5	4.9
eu-2005	112	7.1
in-2004	141	7.8

4.1.5.2 Uniformly Random Sampling

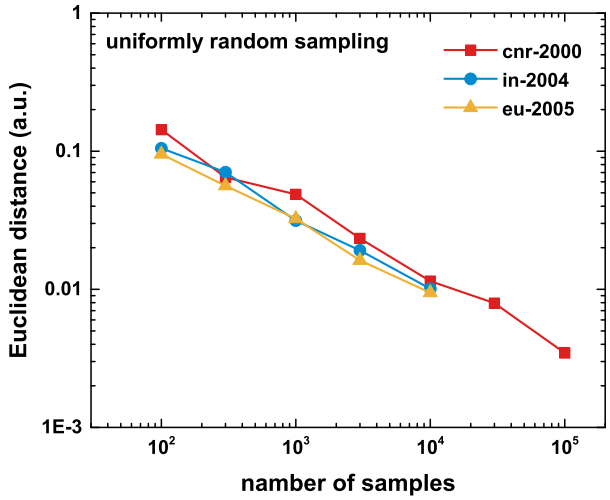

Figure 4.5: Euclidean distance of betweenness centrality between the estimation by uniformly random sampling and the exact computation as a function of the number of samples.

Fig. 4.5 shows the Euclidean distance of betweenness centrality between the estimation by uniformly random sampling and the exact computation as a function of the number of samples. Sampling is without replacement. If the number of samples equals the number of vertices in the graph, the approximation computation becomes the exact computation. The idea of the random sampling is to approximate the betweenness score distribution for all vertices by using a randomly sampled small subset of vertices. More accurate approximation can be achieved by increasing the number of samples. However, the runtime will increase as the number of samples increases. Choosing an appropriate number of samples can achieve both high accuracy and low runtime. We investigate the accuracy as a function of number of samples plotted in

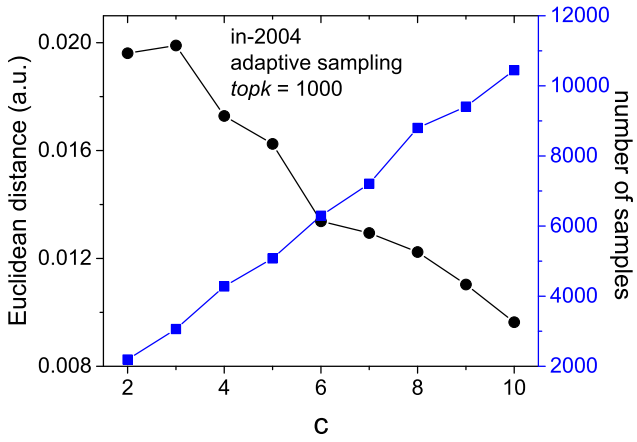

Figure 4.6: Euclidean distance (black curve) of betweenness centrality between the estimation by adaptive sampling and the exact computation as a function of the constant C (required by the adaptive sampling algorithm). The blue curve shows the actual number of samples by adaptive sampling as a function of the constant C.

Fig. 4.5. If we choose the difference of the Euclidean distance below 0.01 to be the criterion of good approximation, the number of samples should be larger than 10^4. The runtime can be reduced to 488 s, 2804 s and 5781 s, for cnr-2000, in-2004 and eu-2005, respectively. The memory consumption for the uniformly random sampling program is the same as the exact computation program.

4.1.5.3 Adaptive Sampling

The adaptive sampling algorithm does not require the predefined number of samples. The algorithm itself determines when to stop the program and the actual number of samples needed. In Fig. 4.6, the black curve shows the Euclidean distance of betweenness centrality between the estimation by adaptive sampling and the exact computation as a function of the constant C for in-2004. The blue curve shows the actual number of samples by adaptive sampling as a function of the constant C. Since the adaptive algorithm does not specify the number of samples to run, constant C balances the approximation accuracy, which is shown by the Euclidean distance difference, with the runtime which is affected by the number of samples to run. Choosing a proper value for C can avoid excessive computation and can still achieve good approximation of betweenness. According to Fig. 4.6, 6 might be a heuristic value for C, with which the actual number of samples needed can be reduced to 6000 with 1728 s runtime and the Euclidean distance difference can still

be maintained around 0.013. The memory consumption for the adaptive sampling program is the same as the exact computation program.

4.2 Diameter

The exact diameter and effective diameter were calculated using the BFS method described in Alg. 8. The BFS method only requires $O(nD)$ space but $O(nm)$ time. For *cnr*-2000, computing the exact diameter and effective diameter took 2.3 h. For larger graphs, using the BFS algorithm to compute the exact diameter and effective diameter is not practical. If we use dynamic programming and use the exact neighborhood function, it requires $O(n^2)$ space, although it only requires $O(nD)$ time. It is still impossible to compute the exact diameter and effective diameter for larger graphs since we cannot afford $O(n^2)$ memory.

Table 4.7: Summary of Diameter Results

Graph	Exact D	Approx. D	error	Exact D_{eff}	Approx. D_{eff}	error
p2p	31	23 ± 3.7	-25.8%	11.75	11.86 ± 0.83	0.9%
slashDot	13	10 ± 0.4	-23.1%	4.81	4.82 ± 0.09	0.2%
cnr-2000	84	78 ± 0.9	-7.1%	25.53	25.41 ± 0.45	-0.5%
ljournal-2008	-	39 ± 1.0	-	-	6.85 ± 0.15	-
eu-2015-*host*	-	25 ± 0.7	-	-	6.85 ± 0.12	-
gsh-2015-*host*	-	23 ± 0.1	-	-	5.71 ± 0.11	-
twitter-2010	-	26	-	-	5.36	-

Table 4.7 compares the diameter estimation results by Alg. 9 using the Flajolet-Martin counters with the exact results by Alg. 8 using BFS. We ran the estimation program 10 times for each dataset and calculated the mean and standard deviation. The results show that Alg. 9 using the Flajolet-Martin counters can give a good estimation for diameter and effective diameter.

Table 4.8 compares the runtime and memory consumption between the estimation program using the Flajolet-Martin counters and the exact computation program using BFS. The time complexity for the exact computation program is too high. Therefore, we did not attempt to compute the exact diameter for graphs larger than cnr-2000. The estimation program has a much less memory footprint due to the Flajolet-Martin counters. With the dynamic programming paradigm, the estimation program can scale to compute graphs with more than one billion edges.

Table 4.8: Runtime and Memory Consumption for Diameter Computation

	Exact		Estimation	
Graph	runtime (s)	memory (GB)	runtime (s)	memory (GB)
p2p	2.2	0.4	1.6	0.1
slashDot	1903	1.6	2.3	0.14
cnr-2000	8260	4.3	15	0.6
ljournal-2008	-	-	200	1.8
eu-2015-*host*	-	-	463	3.3
gsh-2015-*host*	-	-	1042	8.8
twitter-2010	-	-	3908	11.2

4.3 Truss Decomposition

For truss decomposition, we consider undirected simple graphs. Directed graphs are converted to undirected graphs by taking the union with the directed graphs transposed graph. Self-loops in the graph are removed to ensure the simple graph prerequisite. Table 4.9 shows the characteristics of different datasets after removing self-loops. For both the serial and the parallel k-truss decomposition programs, we allocate 4 GB memory for *cnr*-2000, *dblp*-2011, and *ljournal*-2008, 16 GB for *arabic*-2005 and *uk*-2005. For *twitter*-2010, we allocate 22 GB memory for the serial program and 16 GB memory for the parallel program.

Table 4.9: Summary of Datasets after Removing Self-loops

Graph	$\|V\|$	$\|E\|$	t_{max}
cnr-2000	325 557	2 738 969	84
dblp-2011	986 324	3 353 618	119
ljournal-2008	5 363 260	49 514 271	414
arabic-2005	22 744 080	553 903 073	3248
uk-2005	39 459 925	783 027 125	589
twitter-2010	41 652 230	1 202 513 046	1998

4.3.1 Performance Results

Fig. 5.3 shows the runtime of the initial support computation, optimized serial, and optimized parallel k-truss decomposition for different datasets. For small datasets such as *cnr*-2000 and *dblp*-2011, all algorithms can finish in 30 s. For the medium-sized *ljournal*-2008, all algorithms can finish in 15 min. However, for large datasets such as *arabic*-2005, *uk*-2005, and *twitter*-2010, the runtime increases significantly.

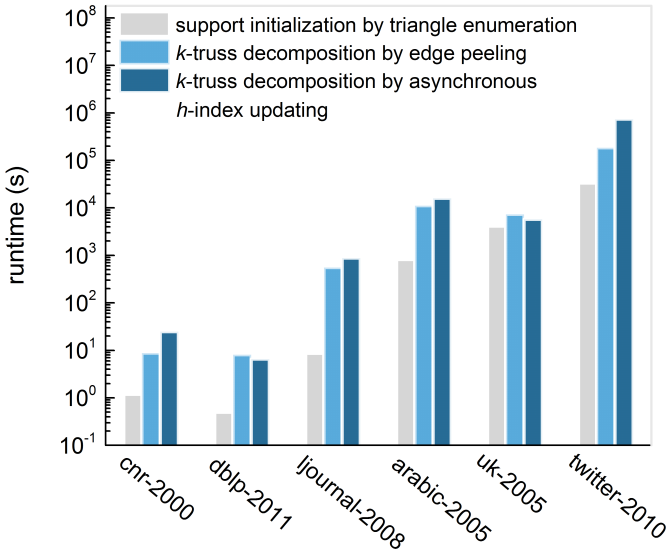

Figure 4.7: Runtime of initial support computation, optimized serial, and optimized parallel k-truss decomposition for different datasets.

Especially for *twitter*-2010, the initial support computation takes 9 h, the serial k-truss decomposition takes 50 h, and the parallel k-truss decomposition takes 203 h. Given the large size of the dataset and the limited computation ability of our machine, the runtime for *twitter*-2010 is still acceptable. We also notice that in most cases, the parallel algorithm even runs slower than the serial algorithm. Since the parallel algorithm processes edges in a random order, it requires many iterations to achieve convergence. By contrast, the serial algorithm processes edges in ascending order of their support. Therefore, it can achieve convergence with only one iteration since it processes each edge for only once. The performance of the parallel algorithm is limited by the CPU of our machine. The parallel algorithm would eventually outperform the serial algorithm on a multi-core server (*e.g.*, 24-core).

Table 4.10: Runtime of Algorithm 11 with Different Implementations

Runtime (s)	*cnr*-2000	*dblp*-2011	*ljournal*-2008
HashMap	1371	47	OutOfMemoryError
Optimized	10	9	555
Speedup ratio	137.1	5.2	–

We implement Algorithm 11 (serial edge peeling) using HashMap in Java standard library[1] as the baseline to compare the performance with our optimized implementa-

[1]https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html

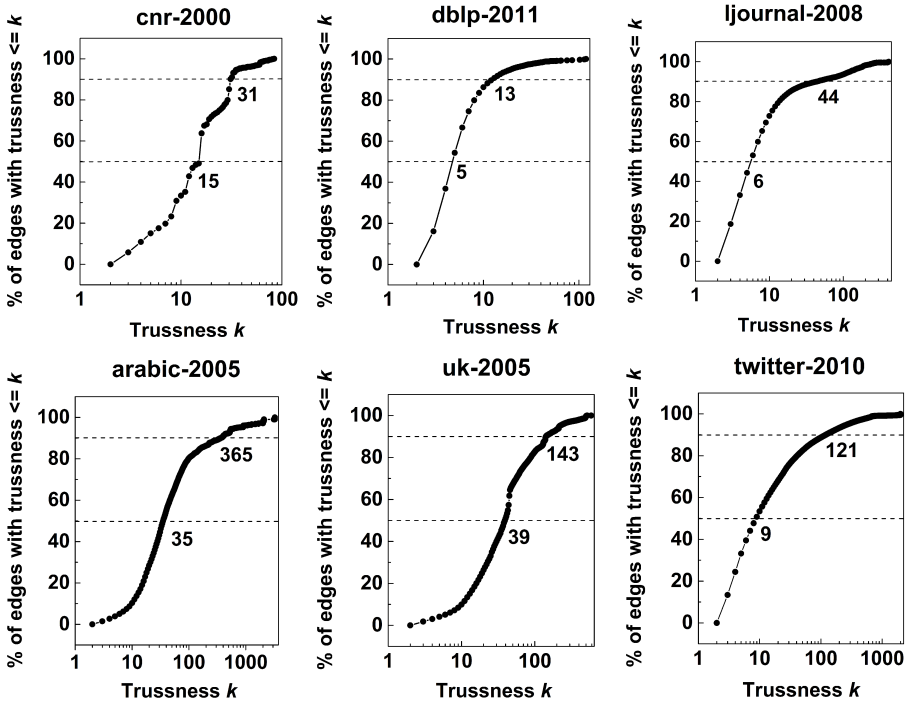

Figure 4.8: Trussness distributions for different datasets.

tion using array-based data structures and WebGraph. Table 4.10 shows the runtime comparison. The HashMap implementation takes 1371 s for *cnr*-2000 and 47 s for *dblp*-2011 while our optimized implementation only takes 10 s for *cnr*-2000 and 9 s for *dblp*-2011, showing a significant speedup. It should be noted that it is the use of array-based structures that contributes to the speedup. The HashMap implementation encounters "OutOfMemoryError" when computing *ljournal*-2008 with the same memory allocation as our optimized implementation (4 GB). The same error occurs when computing larger graphs such as *arabic*-2005, *uk*-2005, or *twitter*-2010 using the HashMap implementation under the same memory allocation condition. Results show the HashMap implementation is neither time nor memory efficient compared to our optimized implementation. We attribute the inefficiency to the increased lookup time in the hash map caused by the potential collisions. Algorithm 12 (asynchronous h-index updating) would suffer the same problem if using the HashMap implementation. Therefore, we did not specifically implement Algorithm 12 using HashMap for performance comparison with our optimized implementation.

Fig. 4.8 shows the trussness distributions for different datasets. The maximum trussness for different datasets are summarized in Table 4.3. The distributions show

that for all datasets, edges with high trussness are only a small percentage of the total edges. The majority of edges have low trussness. For example, in *cnr*-2000 with $t_{max} = 84$, 50% of edges have trussness less than or equal to 15 and 90% of edges have trussness less than or equal to 31.

4.4 Summary

This chapter presents the experimental results using Graph-XLL to compute centrality measures (eigenvector, hub, authority and betweenness), diameter, and truss decomposition for different datasets on a consumer-grade machine. Runtime and memory consumption are recorded. Performance comparison between different graph libraries (igraph, NetworkX and Graph-XLL) will be done in the next chapter.

Chapter 5

Evaluation, Analysis and Comparisons

This chapter compares the performance (runtime and memory consumption) of algorithms computing centrality measures, diameter and core decomposition across different graph libraries (igraph, NetworkX and Graph-XLL). For truss decomposition, neither igraph nor NetworkX provides equivalent programs. Therefore, we investigate two schemes to further reduce the memory consumption and evaluate their performance for truss decomposition programs in Graph-XLL.

5.1 Centrality Measures

5.1.1 Eigenvector, Hub, Authority, and PageRank

Algorithms computing eigenvector, hub, authority and PageRank in Graph-XLL have similar implementations. For simplicity, we use PageRank as the representative for these four centrality algorithms to perform the transverse comparison.

Fig. 5.1 compares the runtime of computing PageRank for different datasets using igrpah, Graph-XLL and NetworkX, respectively. igraph fails to compute eu-2015-host or any larger graphs due to the out-of-memory error. NetworkX fails to compute ljournal-2008 or any larger graphs due to the same error. By contrast, Graph-XLL is able to process extra large graph such as twitter-2010, showing much better scalability than igraph and NetworkX.

Fig. 5.2 compares the memory consumption of computing PageRank for different datasets using igraph, Graph-XLL and NetworkX, respectively. NetworkX shows the

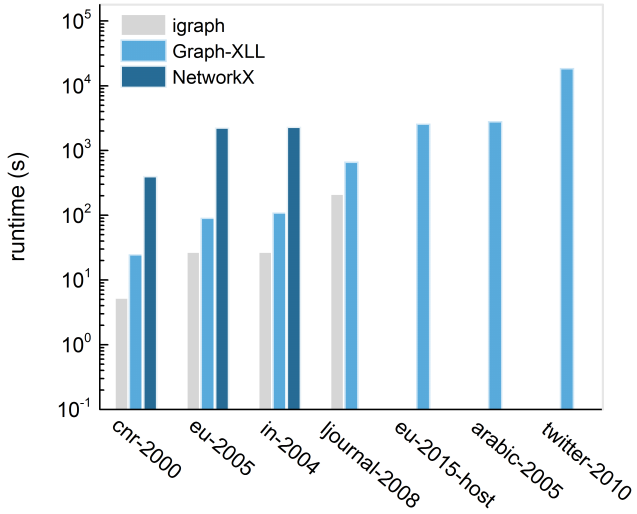

Figure 5.1: Runtime comparison of computing PageRank for different datasets using igraph, Graph-Xll and NetworkX. Graph-XLL is able to process large graphs up to twitter-2010 while the largest graph that igraph can process is ljournal-2008 and in-2004 for NetworkX.

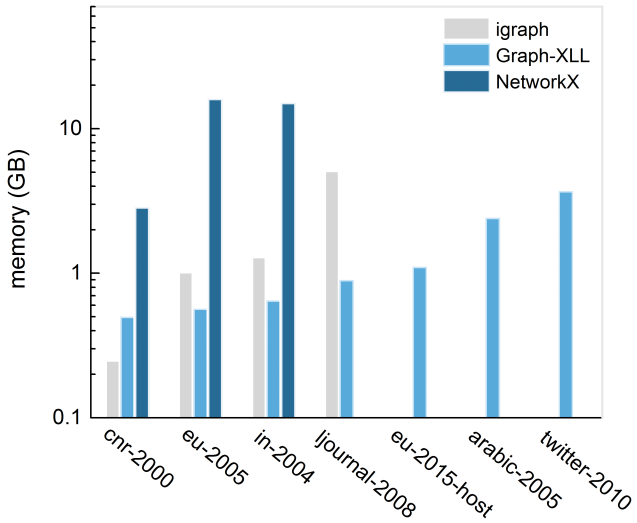

Figure 5.2: Memory consumption comparison of computing PageRank for different datasets using igraph, Graph-Xll and NetworkX. Graph-XLL is able to process large graphs up to twitter-2010 while the largest graph that igraph can process is ljournal-2008 and in-2004 for NetworkX.

worst scalability as the memory consumption is high even for small-sized graphs. The largest graph that NetworkX can process is in-2004. igraph shows better scalability as the slope of the trend is smaller. Even so, igraph fails to process graphs larger than ljournal-2008. The large memory footprint is caused by the graph representation

strategy which fits the complete graph into the memory and the auxiliary data structures used by these two libraries. Fitting the complete graph into the memory would inevitably increase the memory footprint as the size of the graph grows, limiting the scalability of igraph and NetworkX. Besides the graph representation strategy, the data structures used also affect the memory footprint. For example, NetworkX uses hash maps (dictionaries in Python) for graph representation. The large overhead of the auxiliary data structures would increase the memory footprint significantly as well. Graph-XLL shows the best scalability as the slope of the trend is the smallest. For computing twitter-2010, the largest graph in the datasets, Graph-XLL only needs 4 GB memory. Compared to the size of twitter-2010 (the edge list file is over 20 GB), Graph-XLL shows great memory efficiency for processing such large graphs.

5.1.2 Betweenness

Table 5.1: Runtime and Memory Consumption for Exact Betweenness

Graph	Graph-XLL	igraph	NetworkX
`cnr-2000`	5 h, 4.9 GB	3 h, 0.3 GB	not finish in 24 h, 2.9 GB
`eu-2005`	112 h, 7.1 GB	82 h, 1.1 GB	-
`in-2004`	141 h, 7.8 GB	103 h, 1.3 GB	-

For betweenness centrality, igraph and NetworkX only implement the exact computation. Therefore, we compare the exact computation algorithm across igraph, NetworkX and Graph-XLL. Table 5.1 compares the runtime and memory consumption for exact betweenness computation across igraph, NetworkX and Graph-XLL. We can see that none of these exact computation algorithms scales. The scalability is limited by the large time complexity of computing the exact betweenness. Therefore, it is not feasible to use these programs to compute betweenness for large datasets. It is interesting to note that for the betweenness algorithm, Graph-XLL shows larger memory consumption than igraph and NetworkX, which is caused by the extra memory consumption by the parallel process in Graph-XLL and by the reason that many resources are not shared among threads (e.g., queues for BFS), while programs in igraph and NetworkX are sequential.

In Graph-XLL, we implement two betweenness estimation algorithms (uniformly random sampling and adaptive sampling). Neither igraph nor NetworkX provides equivalent programs. The estimation algorithms in Graph-XLL show much less run-

time compared to the exact computation algorithm. For example, the adaptive sampling only takes 1728 s for in-2004, compared to 103 h using igraph.

5.2 Diameter

Similar to the betweenness program, diameter program is also compute-intensive. igraph and NetworkX only provide the exact computation programs. These programs do not scale to large graphs due to the extremely high time complexity. By contrast, in Graph-XLL, we implement an estimation program using the Flajolet-Martin counters. Table 5.2 compares the runtime and memory consumption across Graph-XLL, igraph and NetworkX. Results show that the estimation algorithm in Graph-XLL has much better performance and scalablity than the exact computation programs in igraph and NetworkX.

Table 5.2: Runtime and Memory Consumption for Diameter Computation

Graph	Graph-XLL	igraph	NetworkX
p2p	1.6 s, 0.1 GB	0.4 s, 0.1 GB	46 s, 0.1 GB
slashDot	2.3 s, 0.14 GB	314 s, 0.2 GB	37,306 s, 0.7 GB
cnr-2000	15 s, 0.6 GB	1454 s, 0.3 GB	not finish in 24 h, 2.8 GB
ljournal-2008	200 s, 1.8 GB	not finish in 24 h, 5.2 GB	-
eu-2015-*host*	463 s, 3.3 GB	-	-
gsh-2015-*host*	1042 s, 8.8 GB	-	-
twitter-2010	3908 s, 11.2 GB	-	-

5.3 Core Decomposition

k-core decomposition is a well-established metric which partitions a graph into layers from external to more central vertices. Our previous work [11] showed a vertex-centric implementation for k-core decomposition with great memory efficiency. Both igraph and NetworkX provide an $O(m)$ program based on [42]. We compare the performance of the k-core decomposition program across Graph-XLL, igraph and NetworkX.

Fig. 5.3 compares the runtime of computing k-core for different datasets using igrpah, Graph-XLL and NetworkX, respectively. igraph and NetworkX failed to compute eu-2015-host or any larger graphs due to the out-of-memory error. By contrast, Graph-XLL is able to process extra large graph such as twitter-2010, showing much better scalability than igraph and NetworkX.

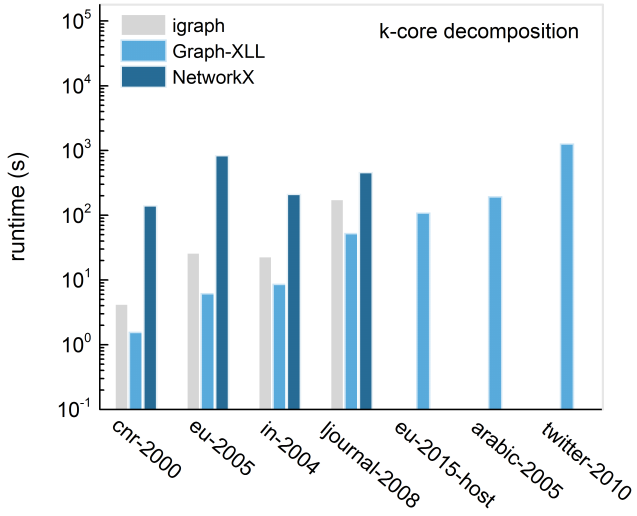

Figure 5.3: Runtime comparison of computing k-core for different datasets using igraph, Graph-Xll and NetworkX. Graph-XLL is able to process large graphs up to twitter-2010 while the largest graph that igraph and NetworkX can process is ljournal-2008.

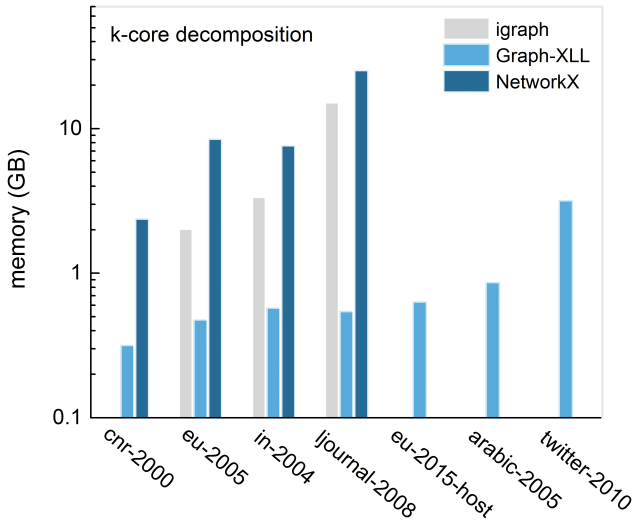

Figure 5.4: Memory consumption comparison of computing k-core for different datasets using igraph, Graph-Xll and NetworkX. Graph-XLL is able to process large graphs up to twitter-2010 while the largest graph that igraph and NetworkX can process is ljournal-2008.

Fig. 5.4 compares the memory consumption of computing k-core for different datasets using igraph, Graph-XLL and NetworkX, respectively. igraph and NetworkX failed to compute eu-2015-host or any larger graphs due to the out-of-memory error. By contrast, Graph-XLL shows much better scalability. For computing twitter-2010,

the largest graph in the datasets, Graph-XLL only needs less than 4 GB memory. Compared to the size of twitter-2010 (the edge list file is over 20 GB), Graph-XLL shows great memory efficiency for processing such large graphs.

5.4 Truss Decomposition

In this section, we investigate two schemes to further reduce the memory usage and evaluate their performance. One is to remove the *edgeTail* array. The other is to remove the *edgePos* array. Both schemes, described in detail, below can reduce the memory usage from $(4m + 2n)$ to $(3m + 2n)$ for Algorithm 11 and from $(3m + 2n)$ to $(2m + 2n)$ for Algorithm 12. However, as we will show, both schemes would result in a significant performance drop for the runtime and scalability. This suggests that the data structures proposed in 3.3.4 have the minimum memory requirement for Algorithms 11 and 12 in order to have good performance, and these are $(4m + 2n)$ for Algorithm 11 and $(3m + 2n)$ for Algorithm 12.

The first scheme is to remove *edgeTail* array shown in Fig. 3.2. We can use Webgraph's API to export *edgeTail* to a new compressed graph in which each vertex only has neighbors with larger vertex ID. We implement Algorithm 11 using this scheme. Results show that removing *edgeTail*, the runtime for the k-truss decomposition of *cnr*-2000 is significantly increased from 10 s to 1735 s. We do not attempt to run this implementation on other larger graphs. Removing *edgeTail* would affect both procedures of checking the existence of an edge and getting the edge ID since both procedures depend on the binary search in the neighbor sets contained in *edgeTail*. Without *edgeTail* array, each binary search operation requires random access to the compressed graph (including the decompression from the compressed graph and disk I/O), which is very time consuming. Therefore, removing *edgeTail* is not of practical use, which also applies to Algorithm 12.

The other scheme is to remove the *edgePos* array shown in Fig. 3.2. *edgePos* is used to store the position of an edge in *sortedEdge* array (see the explanation in 3.3.4). If we remove *edgePos*, we have to search for a given edge in *sortedEdge* since we lose the convenience of getting the edge position directly from *edgePos*. Since the binary search operation requires a sorted array, we have to maintain *sortedEdge* sorted with regard to the edge ID in each support segment. Each time when the support of an edge is decreased, we have to move this edge to the corresponding support segment in *sortedEdge* and insert the edge to the right position in order to

maintain the support segment sorted. The insertion operation involves moving a large amount of elements in *sortedEdge*, which would increase the runtime. We implement Algorithm 11 using this scheme. Results show that the k-truss decomposition of *cnr*-2000 takes 145 s, much faster than the first scheme (1735 s) but still 14× slower than our optimized implementation (10 s). For larger graphs such as *ljournal*-2008, the performance becomes worse as the truss decomposition takes 37496 s (67 × slower than the optimized implementation of 555 s), showing poor scalability.

In short, results show that any further memory reduction would cause significant degradation on the runtime and scalability of the k-truss decomposition programs, which proves that our carefully engineered data structures ($(4m+2n)$ for Algorithm 11 and $(3m+2n)$ for Algorithm 12) are indeed the optimum design to be both time and memory efficient.

5.5 Summary

We performed transverse comparison of algorithms computing centrality, diameter and k-core among Graph-XLL, igraph and NetworkX in terms of runtime and memory consumption to investigate the scalability. Results show that programs in igraph and NetworkX do not scale. For algorithms computing eigenvector, hub, authority, PageRank and k-core in igraph and NetworkX, the main constraint for scalability is the memory since igraph and NetworkX require the complete graph should fit in the main memory, which undoubtedly hinders the scalability to process large graphs. For algorithms computing betweenness and diameter in igraph and NetworkX, the main constraint for scalablility is the large time complexity of the algorithms performing exact computation.

To overcome the time and space constraints, in Graph-XLL, we implement the algorithms using the WebGraph framework and the approximate computation. With WebGraph, we carefully engineer the data structures used in the algorithms to minimize the memory footprint, which enables Graph-XLL to process extra large graphs with more than one billion edges within a reasonable amount of time. To overcome the large time-complexity constraint, in Graph-XLL, we provide algorithms for approximate computation (betweenness estimation using uniformly random sampling and adaptive sampling, diameter estimation using the Flajolet-Martin counters). The approximate algorithms can reduce the runtime significantly, which facilitates the scalability to process large graphs.

Chapter 6

Conclusions

In this thesis, we presented our implementations of various graph algorithms computing centrality measures (eigenvector, hub, authority, PageRank and betweenness), diameter and truss decomposition as a graph library, Graph-XLL, with the focus on the scalability. We showed that Graph-XLL can efficiently process extra large graphs with more than one billion edges on a single consumer-grade machine. Other existing graph libraries designed for single machine such as igraph and NetworkX cannot process computations of such scale, thus demonstrating significantly better scalability of Graph-XLL.

Graph-XLL also contains algorithms which are popular in graph analytics but cannot find corresponding implementations in igraph or NetworkX, for example, the truss decomposition algorithm. In Graph-XLL, we engineered two k-truss decomposition algorithms. We showed that the use of a hash table suggested by the original paper is both time and memory consuming in the practical implementation. We optimized the data structure design by using array-based structures. We also designed corresponding operations on the array-based structures to achieve the same functionality as a hash table but with a much smaller memory footprint and better performance. We showed that it is viable to compute the k-truss decomposition of large graphs (up to 1.2 billion edges) on a consumer-grade machine within a reasonable amount of time.

On the other hand, Graph-XLL still misses a few algorithms for computing analytics present in igraph and NetworkX, such as closeness and cliques. Devising scalable algorithms for computing the closeness and cliques will be part of our future work.

Bibliography

[1] H. Bast, E. Carlsson, A. Eigenwillig, R. Geisberger, C. Harrelson, V. Raychev, and F. Viger, "Fast routing in very large public transportation networks using transfer patterns," in *European Symposium on Algorithms*, pp. 290–301, Springer, 2010.

[2] E. Bullmore and O. Sporns, "Complex brain networks: graph theoretical analysis of structural and functional systems," *Nature reviews neuroscience*, vol. 10, no. 3, p. 186, 2009.

[3] M. V. Marathe and A. K. S. Vullikanti, "Computational epidemiology," in *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 1969–1969, ACM, 2014.

[4] A. Mislove, M. Marcon, K. P. Gummadi, P. Druschel, and B. Bhattacharjee, "Measurement and analysis of online social networks," in *Proceedings of the 7th ACM SIGCOMM conference on Internet measurement*, pp. 29–42, ACM, 2007.

[5] S. Sahu, A. Mhedhbi, S. Salihoglu, J. Lin, and M. T. Özsu, "The ubiquity of large graphs and surprising challenges of graph processing," *Proceedings of the VLDB Endowment*, vol. 11, no. 4, pp. 420–431, 2017.

[6] G. Csardi, T. Nepusz, *et al.*, "The igraph software package for complex network research," *InterJournal, Complex Systems*, vol. 1695, no. 5, pp. 1–9, 2006.

[7] A. Hagberg, P. Swart, and D. S Chult, "Exploring network structure, dynamics, and function using networkx," tech. rep., Los Alamos National Lab.(LANL), Los Alamos, NM (United States), 2008.

[8] P. Boldi and S. Vigna, "The webgraph framework i: compression techniques," in *Proceedings of the 13th international conference on World Wide Web*, pp. 595–602, ACM, 2004.

[9] J. Lu and A. Thomo, "An experimental evaluation of giraph and graphchi," in *2016 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM)*, pp. 993–996, IEEE, 2016.

[10] Y. Santoso, A. Thomo, V. Srinivasan, and S. Chester, "Triad enumeration at trillion-scale using a single commodity machine," in *Advances in Database Technology-EDBT 2019, 22nd International Conference on Extending Database Technology, Lisboa, Portugal, March 26-29, Proceedings*, OpenProceedings. org, 2019.

[11] W. Khaouid, M. Barsky, V. Srinivasan, and A. Thomo, "K-core decomposition of large networks on a single pc," *Proceedings of the VLDB Endowment*, vol. 9, no. 1, pp. 13–23, 2015.

[12] M. Simpson, V. Srinivasan, and A. Thomo, "Efficient computation of feedback arc set at web-scale," *Proceedings of the VLDB Endowment*, vol. 10, no. 3, pp. 133–144, 2016.

[13] M. Simpson, V. Srinivasan, and A. Thomo, "Clearing contamination in large networks," *IEEE Transactions on Knowledge and Data Engineering*, vol. 28, no. 6, pp. 1435–1448, 2016.

[14] D. Popova, A. Khot, and A. Thomo, "Data structures for efficient computation of influence maximization and influence estimation.," in *EDBT*, pp. 505–508, 2018.

[15] D. Popova, N. Ohsaka, K.-i. Kawarabayashi, and A. Thomo, "Nosingles: a space-efficient algorithm for influence maximization," in *Proceedings of the 30th International Conference on Scientific and Statistical Database Management*, p. 18, ACM, 2018.

[16] S. Chen, R. Wei, D. Popova, and A. Thomo, "Efficient computation of importance based communities in web-scale networks using a single machine," in *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*, pp. 1553–1562, ACM, 2016.

[17] C. B. Weinstock and J. Goodenough, "On system scalability," 2006.

[18] P. Bonacich, "Some unique properties of eigenvector centrality," *Social networks*, vol. 29, no. 4, pp. 555–564, 2007.

[19] J. M. Kleinberg, "Hubs, authorities, and communities," *ACM computing surveys (CSUR)*, vol. 31, no. 4es, p. 5, 1999.

[20] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: Bringing order to the web.," tech. rep., Stanford InfoLab, 1999.

[21] M. Barthelemy, "Betweenness centrality in large complex networks," *The European physical journal B*, vol. 38, no. 2, pp. 163–168, 2004.

[22] U. Brandes, "A faster algorithm for betweenness centrality," *Journal of mathematical sociology*, vol. 25, no. 2, pp. 163–177, 2001.

[23] D. A. Bader, S. Kintali, K. Madduri, and M. Mihail, "Approximating betweenness centrality," in *International Workshop on Algorithms and Models for the Web-Graph*, pp. 124–137, Springer, 2007.

[24] S. Ji and Z. Yan, "Refining approximating betweenness centrality based on samplings," *arXiv preprint arXiv:1608.04472*, 2016.

[25] J. Guare, *Six degrees of separation: A play*. Vintage, 1990.

[26] P. Flajolet and G. N. Martin, "Probabilistic counting algorithms for data base applications," *Journal of computer and system sciences*, vol. 31, no. 2, pp. 182–209, 1985.

[27] U. Kang, C. E. Tsourakakis, A. P. Appel, C. Faloutsos, and J. Leskovec, "Radius plots for mining tera-byte scale graphs: Algorithms, patterns, and observations," in *Proceedings of the 2010 SIAM International Conference on Data Mining*, pp. 548–558, SIAM, 2010.

[28] G. Zhao and J. Yuan, "Discovering thematic patterns in videos via cohesive subgraph mining," in *2011 IEEE 11th International Conference on Data Mining*, pp. 1260–1265, IEEE, 2011.

[29] Y. Shao, L. Chen, and B. Cui, "Efficient cohesive subgraphs detection in parallel," in *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, pp. 613–624, ACM, 2014.

[30] N. Wang, S. Parthasarathy, K.-L. Tan, and A. K. Tung, "Csv: visualizing and mining cohesive subgraphs," in *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pp. 445–458, ACM, 2008.

[31] F. Moser, R. Colak, A. Rafiey, and M. Ester, "Mining cohesive patterns from graphs with feature vectors," in *Proceedings of the 2009 SIAM International Conference on Data Mining*, pp. 593–604, SIAM, 2009.

[32] J. Cohen, "Trusses: Cohesive subgraphs for social network analysis," *National security agency technical report*, vol. 16, pp. 3–1, 2008.

[33] X. Huang, H. Cheng, L. Qin, W. Tian, and J. X. Yu, "Querying k-truss community in large and dynamic graphs," in *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pp. 1311–1322, ACM, 2014.

[34] P.-L. Chen, C.-K. Chou, and M.-S. Chen, "Distributed algorithms for k-truss decomposition," in *2014 IEEE International Conference on Big Data (Big Data)*, pp. 471–480, IEEE, 2014.

[35] X. Huang, W. Lu, and L. V. Lakshmanan, "Truss decomposition of probabilistic graphs: Semantics and algorithms," in *Proceedings of the 2016 International Conference on Management of Data*, pp. 77–90, ACM, 2016.

[36] A. M. Katunka, C. Yan, K. B. Serge, and Z. Zhang, "K-truss based top-communities search in large graphs," in *2017 Fifth International Conference on Advanced Cloud and Big Data (CBD)*, pp. 244–249, IEEE, 2017.

[37] C. Bron and J. Kerbosch, "Algorithm 457: finding all cliques of an undirected graph," *Communications of the ACM*, vol. 16, no. 9, pp. 575–577, 1973.

[38] F. Zhao and A. K. Tung, "Large scale cohesive subgraphs discovery for social network visual analysis," *Proceedings of the VLDB Endowment*, vol. 6, no. 2, pp. 85–96, 2012.

[39] X. Huang, L. V. Lakshmanan, J. X. Yu, and H. Cheng, "Approximate closest community search in networks," *Proceedings of the VLDB Endowment*, vol. 9, no. 4, pp. 276–287, 2015.

[40] A. Verma, A. Buchanan, and S. Butenko, "Solving the maximum clique and vertex coloring problems on very large sparse networks," *INFORMS Journal on computing*, vol. 27, no. 1, pp. 164–177, 2015.

[41] J. Wang and J. Cheng, "Truss decomposition in massive networks," *Proceedings of the VLDB Endowment*, vol. 5, no. 9, pp. 812–823, 2012.

[42] V. Batagelj and M. Zaversnik, "An o (m) algorithm for cores decomposition of networks," *arXiv preprint cs/0310049*, 2003.

[43] X. Hu, F. Liu, V. Srinivasan, and A. Thomo, "k-core decomposition on giraph and graphchi," in *International Conference on Intelligent Networking and Collaborative Systems*, pp. 274–284, Springer, 2017.

[44] B. Tootoonchi, V. Srinivasan, and A. Thomo, "Efficient implementation of anchored 2-core algorithm," in *Proceedings of the 2017 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining 2017*, pp. 1009–1016, ACM, 2017.

[45] H. Zhang, H. Zhao, W. Cai, M. Zhao, and G. Luo, "Visualization and cognition of large-scale software structure using the k-core analysis," in *2008 International Conference on Intelligent Information Hiding and Multimedia Signal Processing*, pp. 954–957, IEEE, 2008.

[46] H. Kabir and K. Madduri, "Shared-memory graph truss decomposition," in *2017 IEEE 24th International Conference on High Performance Computing (HiPC)*, pp. 13–22, IEEE, 2017.

[47] S. Smith, X. Liu, N. K. Ahmed, A. S. Tom, F. Petrini, and G. Karypis, "Truss decomposition on shared-memory parallel systems," in *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, pp. 1–6, IEEE, 2017.

[48] A. E. Sarıyüce, C. Seshadhri, and A. Pinar, "Parallel local algorithms for core, truss, and nucleus decompositions," *arXiv. org e-Print archive, https://arxiv. org/abs/1704.00386*, 2017.

[49] C. Voegele, Y.-S. Lu, S. Pai, and K. Pingali, "Parallel triangle counting and k-truss identification using graph-centric methods," in *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, pp. 1–7, IEEE, 2017.

[50] S. Samsi, V. Gadepally, M. Hurley, M. Jones, E. Kao, S. Mohindra, P. Monticciolo, A. Reuther, S. Smith, W. Song, *et al.*, "Graphchallenge. org: Raising the bar on graph analytic performance," in *2018 IEEE High Performance extreme Computing Conference (HPEC)*, pp. 1–7, IEEE, 2018.