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

Jian Wu, Venkatesh Srinivasan, Alex Thomo
Department of Computer Science, University of Victoria
Victoria, BC, V8P 5C2, Canada
{wujian, srinivas, thomo}@uvic.ca

***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 paper, we introduce 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.**

***Index Terms*—graph analytics, scalability, centrality**

I. Introduction

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 and the NetworkX Python library are some of the most popular existing graph libraries due to their easy-to-use off-the-shelf feature [6], [7]. 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 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 paper, we introduce 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 for the underlying graph representation. WebGraph is a highly efficient graph compression framework [8]. 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 we have implemented a multitude of algorithms in Graph-XLL, we focus in this paper on graph centrality measures. Namely, we showcase our implementations for eigenvector, hub, authority, PageRank, and betweenness centralities using the vertex-centric model. Other scalable algorithms that we have implemented in Graph-XLL compute triad-enumeration, core-decomposition, truss-decomposition, feedback-arc-set, influential-users, and importance-based-communities. There are no algorithms yet implemented for

the last four concepts in igraph or NetworkX despite them being immensely popular concepts in graph analytics (cf. [10]–[15]). For the description of our scalable algorithms for these concepts, we refer the reader to [16]–[25]. On the other hand, Graph-XLL still misses a few algorithms for computing the diameter of the graph, cliques, and closeness. While igraph and NetworkX have algorithms for them, they are not scalable. The quest for scalable algorithms for computing the diameter, cliques, and closeness is part of our future work.

The contributions of this work are summarized as follows:

- We implement various graph algorithms for centrality analysis (e.g., eigenvector, hub and authority, PageRank and betweenness) with the emphasis on the scalability to achieve extra large graph processing up to tens of millions of vertices and billions of edges.
- We perform a thorough experimental study to investigate the scalability using different datasets and compare our implementation with igraph and NetworkX in terms of performance and memory consumption.
- We show that our implementation is capable of efficiently analyzing extra large graphs on a single consumer-grade machine.

II. Algorithms Implementation

This section describes the algorithms implemented in Graph-XLL. We focus on centrality algorithms since they are the most commonly used algorithms for graph analytics. We implement the algorithms in a vertex-centric manner. Computations are broken down to vertex level and are performed in parallel.

A. Eigenvector, Hub, Authority and PageRank

The eigenvector centrality [26] is defined as a measure of the influence of a vertex in a network.

Definition 1: **Eigenvector Centrality** (EC) of a vertex v_i in a graph $G = (V, E)$ is defined as

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

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

Similar measures of the influence of a vertex includes hub centrality [27], authority centrality [27], and PageRank [28], which are initially used to rate the importance of web pages. The mathematical definitions are shown below.

Definition 2: **Hub Centrality** (HC) of a vertex v_i in a graph $G = (V, E)$ is defined as

$$HC(v_i) = \frac{1}{\lambda} \sum_{v_j \in ON(v_i)} \sum_{v_k \in IN(v_j)} HC(v_k), \tag{2}$$

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 1 PageRank Compute Function

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

Definition 3: **Authority Centrality** (AC) of a vertex v_i in a graph $G = (V, E)$ is defined as

$$AC(v_i) = \frac{1}{\lambda} \sum_{v_j \in IN(v_i)} \sum_{v_k \in ON(v_j)} AC(v_k), \tag{3}$$

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.

Definition 4: **PageRank** (PR) of a vertex v_i in a graph $G = (V, E)$ is defined as

$$PR(v_i) = \frac{1-d}{n} + d \sum_{v_j \in IN(v_i)} \frac{PR(v_j)}{D(v_j)}, \tag{4}$$

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 D(v_j) is the out-degree for v_j.

The above equations show the intrinsic vertex-centric feature. The value of a vertex is only influenced by its neighbors close to it. We present the pseudocode for PageRank computation in Alg. 1. Other centrality computations are similar to PageRank computation. All vertices are initialized with the value of $1/n$ at superstep 0. n is the total number of vertices in the graph. For the subsequent supersteps, each vertex will sum all the messages (value divided by out-degree) received from its neighbours and update its value by Eq. 4. We maintain two arrays to record the vertex values for the previous and current supersteps in the program. We first update the current vertex value 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 value using the current value, which will be read for the next superstep. The program stops if the residual is below the predefined tolerance.

B. Betweenness Centrality

The vertex-centric implementation of betweenness centrality is more involved and as such we need a more detailed

description of its components. To the best of our knowledge, there is no work that describes the details of a VC algorithm for computing betweenness centrality.

Betweenness centrality [29] is a measure of a vertex's centrality based on shortest paths. It represents, for each vertex, the number of shortest paths passing through the vertex. The formal definition is shown below.

Definition 5: **Betweenness Centrality** (BC) of a vertex v in a graph $G = (V, E)$ is

$$BC(v) = \sum_{s,t \in V, s \neq t \neq v} \delta_{st}(v), \tag{5}$$

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

$$\delta_{st}(v) = \frac{\sigma_{st}(v)}{\sigma_{st}}, \tag{6}$$

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.

We use Brandes' algorithm [30] to compute betweenness centrality. Brandes' algorithm is currently the fastest algorithm for exact betweenness computation, which is based on the **dependency** of a source vertex on a given vertex and can be implemented in a vertex-centric manner.

Definition 6: Given a graph $G = (V, E)$, the **dependency** of a source vertex $s \in V$ on a vertex $v \in V$ is

$$\delta_s(v) = \sum_{t \in V, s \neq t \neq v} \delta_{st}(v). \tag{7}$$

Based on Definition 5, the BC value of vertex v is

$$BC(v) = \sum_{s \neq v} \delta_s(v). \tag{8}$$

The algorithm uses a recursive way to calculate the BC value of v, by introducing *predecessors* of v.

Definition 7: Given an unweighted 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.

$$t \in P_s(v) \Rightarrow (d(s, v) = d(s, t) + 1) \wedge (t, v) \in E,$$

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

Brandes' algorithm is based on the following theorem:

Theorem 1: Given a graph (V, E), for any $s, v \in V$, we have

$$\delta_s(v) = \sum_{t \in V s.t. v \in P_s(t)} \frac{\sigma_{sv}}{\sigma_{st}}(1 + \delta_s(t)). \tag{9}$$

in which σ_{sv} denotes the total number of the shortest paths from s to v and σ_{st} denotes the total number of the shortest paths from s to t.

Brandes' algorithm can be summerized as follows:

1) For each vertex $s \in V$, we calculate the number of shortest paths from s to all the other vertices, i.e., using breadth-first search for unweighted graphs, the running time is bounded by $O(|E| + |V|)$.
2) For each vertex $s \in V$, traverse the vertices in descending order of their distances from s, and accumulate the dependencies by Theorem 1. Each traversal takes $O(|E|)$ time.

Therefore, the time complexity for Brandes' algorithm is $O(|V| \times (|E| + |V|) + |V||E|) = O(|V||E|)$ for unweighted graphs.

Theorem 1 shows that the dependency can be computed in a vertex-centric manner and the betweenness 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 2. Supersteps are delimited by the minimum step distance from the source vertex. For example, 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.

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 [31] and adaptive sampling [32].

1) Uniformly Random Sampling: According to Theorem 1, 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 selected uniformly at random. The pseudocode for uniform random sampling is shown in Alg 3.

2) Adaptive Sampling: Instead of setting the number of pivots k as one of the input parameters in uniformly random sampling algorithm, the adaptive sampling technique determines the actual number of pivots k through each sampling. There is no need to predefine $k's$ value. The basic idea is

Algorithm 2 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
```

Algorithm 3 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. 2)
5:         accumulation (same as Alg. 2)
6:     /* rescaling */
7:     scale ← n/((n − 1) · (n − 2) · k)
8:     for v ∈ V do BC[v] ← BC[v] · scale
```

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

Algorithm 4 Betweenness (Adaptive Sampling)

```
1: function COMPUTE(G)
2:     k ← 0
3:     while k < cutoff do
4:         s ← sampling a vertex as the pivot
5:         single-source shortest-path (same as Alg. 2)
6:         accumulation (same as Alg. 2)
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
```

parameter. We use $cutoff$ to specify the maximum number of pivots. The pseudocode for adaptive sampling is shown in Alg. 4.

In Graph-XLL, we implement Brandes' algorithm for exact betweenness computation, as well as the uniformly random sampling and adaptive sampling algorithms for betweenness approximation.

III. Experimental Setup

We implement algorithms in Graph-XLL in Java 8 and use parallel stream to achieve parallel computation. 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 seven different datasets obtained from http://law.di.unimi.it/datasets.php. Characteristics of the datasets are summarized in Table 1.

TABLE I
SUMMARY OF DATASETS

Graph	$\lvert V \rvert$	$\lvert E \rvert$
cnr-2000	325,557	3,216,152
eu-2005	862,664	19,235,140
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
twitter-2010	41,652,230	1,468,365,182

IV. Main Results

We investigate the scalability of the algorithms in Graph-XLL using different datasets. We select PageRank results as the representative since PageRank shares the same nature as eigenvector, hub and authority centralities.

Fig. 1 shows the Euclidean distance of PageRank between two consecutive iterations after a certain number of iterations for different graphs. The Euclidean distance can be used as the convergence condition. If we choose 10^{-14} as the criterion

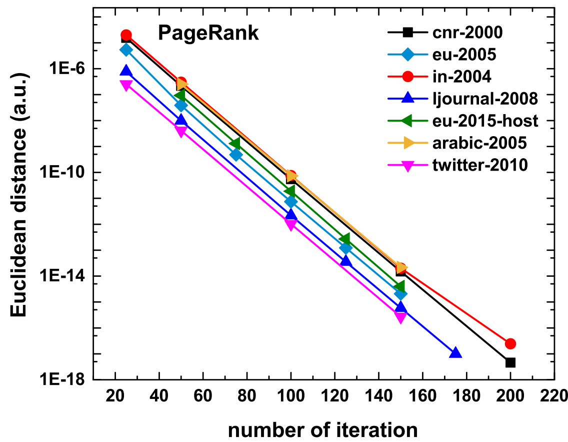

Fig. 1. Euclidean distance of PageRank between two consecutive iterations after a certain number of iterations for different graphs.

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.

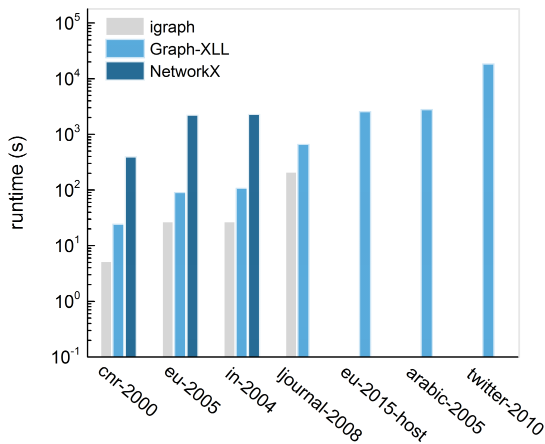

Fig. 2. Runtime comparison of computing PageRank for different datasets using igraph, Graph-Xll and NetworkX. Graph-XLL is able to process large graphs up to twitter-2010 while the largest graph that igraph can process is ljournal-2008 and in-2004 for NetworkX.

Fig. 2 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. 3 compares the memory consumption of computing PageRank for different datasets using igraph, Graph-XLL and NetworkX, respectively. NetworkX shows the 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

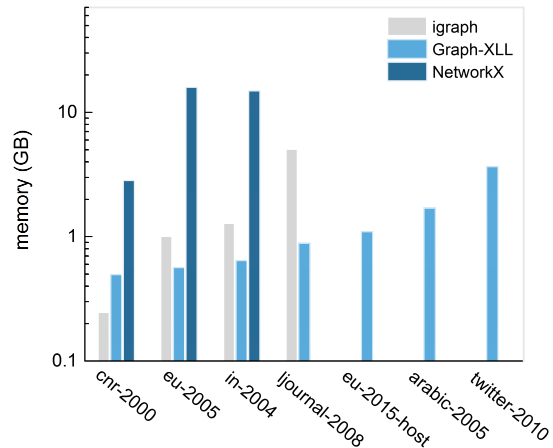

Fig. 3. Memory consumption comparison of computing PageRank for different datasets using igraph, Graph-Xll and NetworkX. Graph-XLL is able to process large graphs up to twitter-2010 while the largest graph that igraph can process is ljournal-2008 and in-2004 for NetworkX.

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. When 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.

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). 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, the smallest graph in the datasets. Both igraph and NetworkX suffer from the same problem. 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.

Fig. 4 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

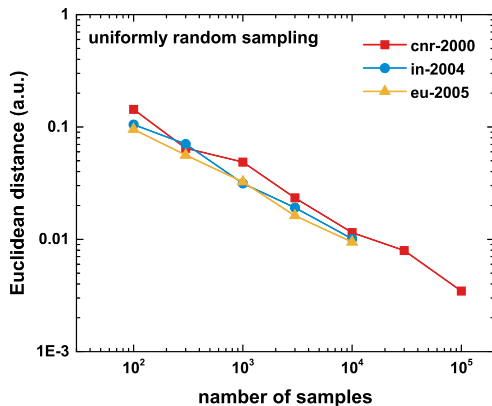

Fig. 4. Euclidean distance of betweenness centrality between the estimation by uniformly random sampling and the exact computation as a function of the number of samples.

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 Fig. 4. 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.

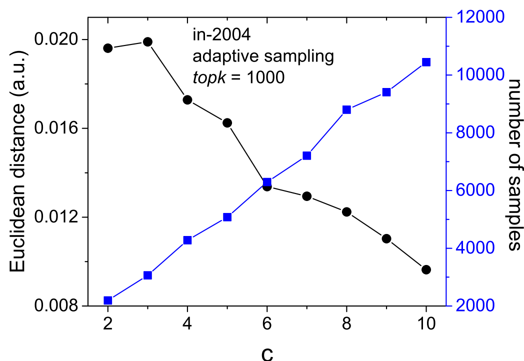

Fig. 5. Euclidean distance (black curve) of betweenness centrality between the estimation by adaptive sampling and the exact computation as a function of the constant C (required by the adaptive sampling algorithm). The blue curve shows the actual number of samples by adaptive sampling as a function of the constant C.

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. 5, 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, 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. 5, 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.

V. Discussion

In terms of scalability, Graph-XLL outperforms igraph and NetworkX by adopting a much better graph representation strategy, which avoids fitting the complete graph into the memory. This strategy equips Graph-XLL with the ability to process extra large graphs with up to ten million vertices and one billion edges (e.g., twitter-2010 with an edge list file of over 20 GB). In order to perform computation, igraph and NetworkX need to load the complete graph into the main memory. With the large overhead of the auxiliary data structures, the actual memory needed by igraph and NetworkX is sometimes multiple times of the size of the graph's edge list file. For example, ljournal-2008 has an edge list file of 1 GB. NetworkX fails to process ljournal-2008 even on a machine with 32 GB memory while igraph needs 5 GB to perform computation. By contrast, Graph-XLL only needs less than 1 GB memory. By extrapolation, if we want to use igraph to process twitter-2010, a machine equipped with at least 100 GB memory should be required, which is still not common for average users. Graph-XLL can scale smoothly with the increasing graph size since the programs only load the needed fraction of the graph into the memory during computation, which decreases the memory footprint significantly.

For compute-intensive algorithms such as betweenness centrality, besides the memory footprint, the runtime of a program may deserve more consideration. Since the exact computation of betweenness centrality is extremely time consuming, we implement the approximation algorithms in Graph-XLL to decrease the runtime. Neither igraph nor networkX provides such approximation algorithms. The idea is to approximate the distribution of betweenness scores for all vertices by only performing computation based on a small fraction of randomly selected vertices. There is a trade-off between the accuracy of approximation and the runtime of the approximation algorithms. Heuristics can be used to achieve both acceptable approximation accuracy and reasonable runtime when executing these approxiamtion algorithms.

VI. Conclusions

We presented our implementations of various graph algorithms for centrality analysis, which can be used off-the-shelf as a graph library, Graph-XLL, with the focus on the scalability. We showed that Graph-XLL can efficiently process extra large graphs with up to tens of millions of vertices and billions of 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. Other scalable algorithms that we have implemented in Graph-XLL compute triad-enumeration, core-decomposition, truss-decomposition, feedback-arc-set, influential-users, and importance-based-communities. There are no algorithms yet implemented for truss-decomposition, feedback-arc-set, influential-users, and importance-based-communities in igraph or NetworkX despite these being immensely popular concepts in graph analytics. On the other hand, Graph-XLL still misses a few algorithms for computing analytics present in igraph and NetworkX, such as diameter of the graph, cliques, and closeness. Devising scalable algorithms for computing the diameter, cliques, and closeness is part of our future work.

REFERENCES

[1] 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*. ACM, 2007, pp. 29–42.

[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] 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*. Springer, 2010, pp. 290–301.

[4] 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*. ACM, 2014, pp. 1969–1969.

[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," Los Alamos National Lab.(LANL), Los Alamos, NM (United States), Tech. Rep., 2008.

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

[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)*. IEEE, 2016, pp. 993–996.

[10] V. Batagelj and A. Mrvar, "A subquadratic triad census algorithm for large sparse networks with small maximum degree," *Social networks*, vol. 23, no. 3, pp. 237–243, 2001.

[11] J. I. Alvarez-Hamelin, L. Dall'Asta, A. Barrat, and A. Vespignani, "Large scale networks fingerprinting and visualization using the k-core decomposition," in *Advances in neural information processing systems*, 2006, pp. 41–50.

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

[13] P. Charbit, S. Thomassé, and A. Yeo, "The minimum feedback arc set problem is np-hard for tournaments," *Combinatorics, Probability and Computing*, vol. 16, no. 1, pp. 1–4, 2007.

[14] D. Kempe, J. Kleinberg, and É. Tardos, "Maximizing the spread of influence through a social network," in *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2003, pp. 137–146.

[15] R.-H. Li, L. Qin, J. X. Yu, and R. Mao, "Influential community search in large networks," *Proceedings of the VLDB Endowment*, vol. 8, no. 5, pp. 509–520, 2015.

[16] Y. Santoso, A. Thomo, V. Srinivasan, and S. Chester, "Triad enumeration at trillion-scale using a single commodity machine," in *EDBT*, 2019, pp. 718–721.

[17] 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.

[18] J. Wu, A. Goshulak, V. Srinivasan, and A. Thomo, "K-truss decomposition of large networks on a single consumer-grade machine," in *2018 IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining (ASONAM)*. IEEE, 2018, pp. 873–880.

[19] F. Esfahani, V. Srinivasan, A. Thomo, and K. Wu, "Efficient computation of probabilistic core decomposition at web-scale," in *EDBT*, 2019, pp. 325–336.

[20] F. Esfahani, J. Wu, V. Srinivasan, A. Thomo, and K. Wu, "Fast truss decomposition in large-scale probabilistic graphs," in *EDBT*, 2019, pp. 722–725.

[21] 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.

[22] ——, "Clearing contamination in large networks," *TKDE*, 2016.

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

[24] 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*. ACM, 2018, p. 18.

[25] 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*. ACM, 2016, pp. 1553–1562.

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

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

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

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

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

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

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