Community-based parallel sampling on social network graphs

Amalia Georgoudi[*]
ageorgoudi@csd.auth.gr
Aristotle University of Thessaloniki
Greece

Angelos Moavinis[*]
moavinis@csd.auth.gr
Aristotle University of Thessaloniki
Greece

Styliani Kyrama[*]
kyrastyl@csd.auth.gr
Aristotle University of Thessaloniki
Greece

Abstract

Online Social Networks (OSNs) have attracted interest as more and more users use them every day. The analysis of OSNs graphs can give various and useful information. Due to the fact that social networks are growing day by day and the analysis in the whole graph sometimes is not even possible, it is necessary to study and analyze a smaller graph. A good and representative sample keeping all the properties of the initial graph could be analyzed easier and requires fewer resources, in terms of time and memory.

A lot of different techniques and algorithms have been proposed, in order to combat the difficulties of obtaining a good sample of the original graph, which is most of the times extremely large, as we speak for real-life networks. In this report, we propose a method based on clustering. Instead of sampling the whole graph we split the dataset into clusters in the form of communities to handle each one of them individually. To extract the sampled nodes from each cluster we use a random-walk algorithm.

For the evaluation of our algorithm, we compare sample and initials graph results based on some graph properties, such as average clustering coefficient, transitivity, etc. Despite making the above-mentioned comparison, we have no comparison with the results of other sampling algorithms, in order to test the efficiency of our implementation, in terms of accuracy but also time. Furthermore, even though we implemented a scalable algorithm, that performs random walks in parallel, we did not apply it to bigger datasets, aiming to ascertain in practice its scalability, and its performance in big data.

Keywords: graph sampling, network sampling, distributed sampling, clusters, random walk

1 Introduction

In this era of online social networks and linked data, which are usually represented as **graphs**, there is great need to study and analyze them without processing all of their entities, due to their massive increase in size. Processing the whole graph would be a very time- and resource-consuming procedure. Therefore, **sampling** appears as a solution to this problem.

[*]All authors contributed equally to this research.

1.1 Graph Sampling

In terms of research and particularly in the field of social networks, a sample is a group of entities (e.g. people, objects, or items) connected with different types of relations (e.g. friends, co-authors), which are taken from a larger population for analysis. By extracting a representative subset of the prototype population, graph sampling can maintain the characteristics of the initial graph while reducing its size. This ensures that we can generalize the findings from the research sample to the population as a whole. We obtain a sample of the population for many reasons as it is usually not practical and almost never economical to process and analyze the original dataset.

As we can understand, graph sampling is essential because it provides an inexpensive, in terms of time and resources, and also an efficient solution for social network analysis. Several sampling algorithms have been proposed in previous studies, and all of them have their own pros and cons, which are discussed in more detail below. However, due to the fact that graphs represent complex interconnections, naive sampling can destroy the salient features that constitute the value of the graph data.

As for the graph sampling techniques, we can distinguish three main groups: node, edge and exploration-based approaches.

1.2 Main challenges

The biggest challenge of graph sampling, as anyone can understand from all the above-mentioned, is ***how*** to derive a small, but representative sample out of the millions' or billions' scale size social graph. There are many factors that may complicate this process and are likely to affect the quality of the sample, causing it to deviate from the original graph, regarding its properties. One of them, for example, can be the type of entities we are looking for and how often they appear in the overall population. Despite the fact that in many cases, the target population can be sampled directly, there are some scenarios in which the target population is hidden, e.g. injection drug users in the urban areas. In this latter case, the researchers have to execute certain sampling algorithms on a graph to explore this population group[1].

However, it is equally important to take into account the computational cost of exporting this sample, in order to have a "quality-quantity" balance. Hence, it is essential to choose wisely the sampling algorithm that best addresses the researcher's needs.

1.3 Our Contribution

In this report, we describe the method we proposed, which is mainly based on clustering. We implement a technique based on the use of a fast community detection algorithm, to split data into clusters and sample each cluster individually, instead of performing a sample on the overall population. We try to obtain the most representative nodes from each cluster and discard the others, while keeping the properties of the original graph. To extract these samples we use a random-walk technique, with the length of the walk vary with the properties of the cluster. The results are evaluated based on graph metrics such as average clustering coefficient, degree sequence etc. More details on that are given in section 3.1, while the code for our implentation is publicly available on Github.

1.4 Outline

The rest of this technical report goes as follows. In section 2, we mention some of the works that have been already conducted in the field of graph sampling and inspired us for implemented the algorithm described in section 3.1. Along with the algorithm, we describe briefly the dataset in 3.2, while giving some more technical information in section 3.3 about the tools, frameworks and libraries we used. After describing all the above, we continue with the presentation of the results (section 4), both in terms of comparing the output graph of our algorithm with the original one, to the extend possible. Finally, in section 5, we come to some conclusions about the algorithm we implemented, but also in what ways we could improve and extend it.

2 Related Work

Online Social Networks (OSNs) have attracted interest as more and more users use them every day. The analysis of OSNs graphs can give various information and that's why it's an inevitable need to obtain representative samples of OSNs graphs. In [2] the authors focused on Facebook graph sampling and they compare different techniques based on their ***bias*** and ***convergence***. The results showed that traditional graph traversal techniques did not perform well, while on the other hand, MHRW, RWRW methods performed really well. In order to improve convergence and decrease the duration of crawls, they used ***parallel walks***, each of them started from a different initial node. With parallel walks, even if a crawler got trapped in a region, the sample will not be affected. When sampling using a specific method, according to [6], we should find a balance between **"sampling quality"** and **"sampling quantity"** to have a more representative sample under a cost constraint. For that reason, this study proposes the use of random walk algorithms with the **random skipping** technique. With the random skipping, the crawler would be able to pass from a node, without sampling it.

There are some other methods suggested for the "problem" of sampling a graph. There are techniques based on clustering[1], or sampling of more than one phase or stages[2], approaches using stratified sampling[7], or even approaches that deal with the dynamic nature of real social networks[5], which were not described at all in this technical report.

In [4], the authors highlight the ***importance of parallelism in graph sampling***. The usage and the information one can extract from graphs is increasingly important, yet enough resources are required. Large and high-complexity graphs can easily outgrow computation and memory capacities. Parallel computing could help to meet the requirements and solve the problem. The question is where to introduce parallelism.

Based on [7], a major challenge of sampling on graphs is how to ***smooth the sampling bias between the high degree and low degree nodes***. What they proposed to reduce this bias of the sampling procedure was the concept of approximate degree distribution and devise a stratification strategy. They develop two graph sampling algorithms that combine the node selection method and the stratified strategy. They used k-means to produce the node clusters and they sample these sub-graphs with a range of sampling fraction between 5% and 25%. The results show that their sampling algorithms preserve several graph properties and behave more accurately than other algorithms.

Another significant contribution related to the field of sampling on graphs is presented in [3], in which the authors introduce ***Node2Vec***, an algorithmic framework for learning continuous feature representations for graph nodes in networks. They perform the learning of a mapping of nodes to a low-dimensional space of features that maximizes the likelihood of preserving network node neighborhoods, they define a flexible notion of a node's network neighborhood, propose a biased random walk procedure and they propose a flexible sampling strategy that, in short, interpolates between BFS and DFS.

3 Proposed Algorithm

In this section, we describe in more detail the method we proposed for graph sampling; a method based on clustering and parallel sampling. Firstly, in 3.1, we are going to present the methodology we followed, the pipeline of our implementation, along with the pseudocode of our algorithm. We furthermore provide, in 3.2 and 3.3, some information about the dataset, but also for the tools and libraries used for the implementation.

[1]Cluster Sampling - Wikipedia
[2]Multistage Sampling - Wikipedia

3.1 Methodology

We implemented a technique based on the use of a fast community detection algorithm, the label propagation algorithm, also known as LPA. **LPA**[3] detects the communities using network structure alone as its guide and does not require a pre-defined objective function or prior information about the communities. After splitting the whole dataset into communities, we handle each one individually.

We implemented a *simple random-walk technique* to traverse the nodes, trying to keep some of the most representative nodes of each cluster. More specifically, we first define the vertices and edges of the cluster, discarding those that lead to nodes outside the cluster. Then, for each one of the vertices, we define its intra-cluster neighborhood, i.e. the nodes to which this node has an edge. To start the random walk procedure, we randomly pick a node from the available ones and define the length of the walk based on some criteria, which are described later in this subsection.

Pure clustering sampling techniques operate as follows: after creating clusters on data and assigning each node to the cluster it belongs to, only some of the clusters are used for sampling. Most of the times, cluster sampling techniques use only one or two clusters to perform sampling. Contrary to that, we decided to use all of our available clusters for sampling, and since these clusters are non-overlapping, sampling on each one of them can be conducted ***in parallel***.

The clusters, or communities, that are created consist of a different number of nodes, so the nodes we want to keep from each community can not be the same. Assume we have two clusters, A and B, with a number of nodes 10 and 150 respectively. If we had a fixed number of nodes to keep from each cluster e.g. 50, then cluster A does not meet the requirements, as it consists of only 10 nodes, less than 50.

Another concern was of to what degree the nodes in this sub-graph of the community tend to cluster together. A graph property that reveals the aforementioned tendency is the average clustering coefficient, an alternative for the global clustering coefficient.

Therefore, we decided that the number of nodes to be sampled from a community was dependent on two parameters: i. the number of nodes of the cluster, ii. the average clustering coefficient (acc) multiplied by a factor *alpha*. The equation we used to compute the exact percentage of nodes is defined as:

$$perc = \frac{N}{1 + alpha * ACC}$$

Once performing the random walk on each one of the clusters individually, we join the partial results, and construct the sampled graph, based on the result edges. To evaluate the result of our algorithm we compare the initial graph G, with the sampled graph S, based on some graph properties, such as global clustering coefficient, average degree, transitivity, betweenness and closeness centrality. The results of these comparisons are presented and discussed later, in section 4.

The exact procedure we followed is described in the form of pseudocode by the Algorithm 1 and is also illustrated in the figure 1.

Algorithm 1 Cluster Sampling

```
1: Input G: initial graph
2: Output S: sampled graph
3:
4: vertices_partitions = LPA(G, maxIter=m)
5: groupedRDD = join(partition ids, vertices, edges)
6: random_walks = []
7: for p in partitionIDs do
8:     rw = RANDOM_WALK(partition_data)
9:     random_walks.extend(rw)
10: sampled_v = random_walks                ▷ sampled vertices
11: sampled_e = edges between sampled vertices
12: sampled_g = Graph(sampled_v, sampled_e)
13: return sampled_graph
14:
15: procedure RANDOM_WALK(partition_data)
16:     define vertices and edges from partition_data
17:     define neighborhood for each vertex
18:     set max_len of walk =N/ (1+a*ACC(p)))
19:     visited = []
20:     randomly pick a node n to start the walk
21:     for i=0 until i=max_len do
22:         compute transition probabilities
23:         from n to its neighbors, nbh
24:         pick randomly one of nbh, n', with probability p
25:         append visited with n' if n' ∉ visited
26:     return visited
```

3.2 Dataset

The dataset used for this work is the well-known and widely-used **egonets-Facebook** graph from SNAP which is an **undirected** network that models the friendships among Facebook users and consists of 4039 vertices and 88234 edges. Its average clustering coefficient is 0.6055 and its diameter is 8, while the rest of the graph's properties are presented later, in table 1.

The data of this dataset was collected from survey participants using Facebook app, and the dataset includes metadata for each node-user, along with its connections ("friends"), which are practically the edges of the graph. Due to GDPR, Facebook data has been anonymized by replacing the users' ids with a new value, but also, obscuring feature vectors from this dataset with their interpretation. For instance, where the original dataset may have contained a feature "political : Democratic Party", the new data would simply contain "political : anonymized feature 1". Thus, using the anonymized

[3] LPA - Wikipedia

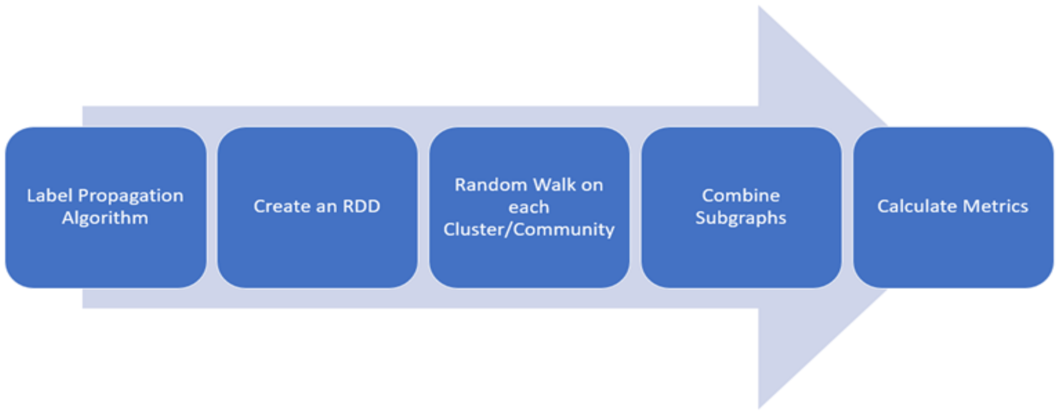

Label Propagation Algorithm
Create an RDD
Random Walk on each Cluster/Community
Combine Subgraphs
Calculate Metrics

Figure 1. Pipeline of Sampling

data it is still possible to determine whether two users have the same political affiliations, but not what their individual political affiliations represent.

The sampling that performed in this work was not at all related to the personal information of users', even though some metadata could have been used to determine the communities in a more content-related way. Thus, the metadata was completely discarded and we only used the information about the structure of the graph, i.e. the set of nodes and their connections.

3.3 Technical details

We executed our experiments on Python and Spark (PySpark, Spark version 3.0.1) on home computers (4-core processors, 8GB RAM). We used the NetworkX library for calculating the graph's related properties, but also visualizing the graph. We also used the GraphFrames library of PySpark to deal with the parallelization of the graph.

PySpark has been released in order to support the collaboration of Apache Spark (originally written in Scala) and Python and it is a Python API for Spark. PySpark offers interaction with RDDs and DataFrames in Apache Spark and Python programming language, which is accomplished by using the Py4j library. PySpark comes with quite a few libraries for writing efficient programs. Furthermore, there are various external libraries that are also compatible. Some examples: PySparkSQL, MLlib and GraphFrames.

GraphFrames is a purpose graph processing library that provides a set of APIs for performing graph analysis efficiently, using the PySpark core and PySparkSQL. It is optimized for fast distributed computing.

NetworkX is a Python library for studying graphs and networks. NetworkX is suitable for operation on large real-world graphs. Due to its dependence on a pure-Python "dictionary of dictionary" data structure, NetworkX is a reasonably efficient, very scalable, highly portable framework for network and social network analysis.

4 Results and Evaluation

4.1 Experiments on maximum iterations

The Label Propagation Algorithm runs for a specified number of maximum iterations and in each iteration it combines some communities into one. Thus for more iterations we get fewer communities, as more little communities will be merged into a bigger one. In plot 2 is depicted the number of communities created for four different values of "max_iter" parameter, the values we used in our own experiments.

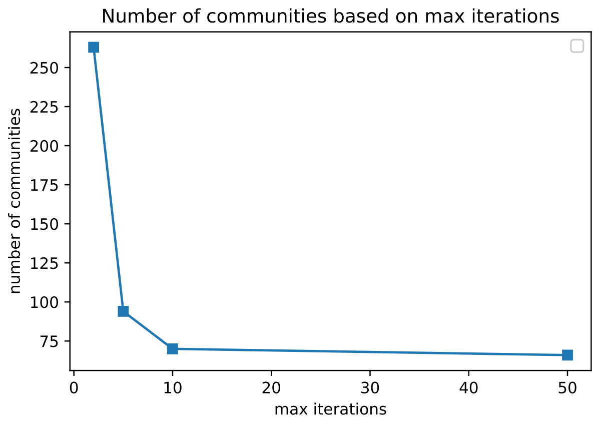


Figure 2. Number of communities

To be able to determine the effect that the number of iterations of the LPA algorithm has on the graph extracted from the algorithm, we visualized the graph for different values of the variable max iter. These visualizations are presented in figure 3, where we can see the original graph and two sampled graphs produced with and maxIter=2 and maxIter=5, while the *alpha* parameter is kept constant at 2.

The first thing one can notice with the help of plots is that we no longer have one single connected graph but many small sub-graphs that correspond to the samples obtained

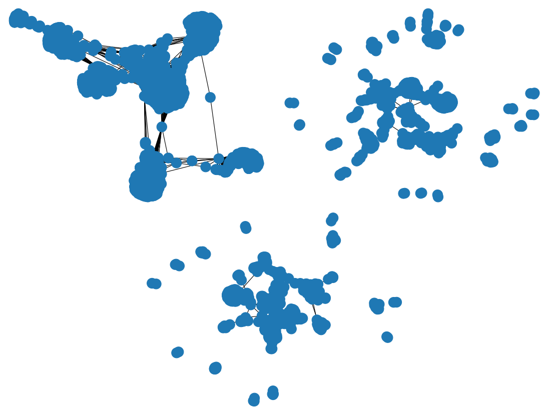


Figure 3. The original graph and its sampled versions, changing maxIter. Top left: original, bottom: maxIter=2, top right: maxIter=5

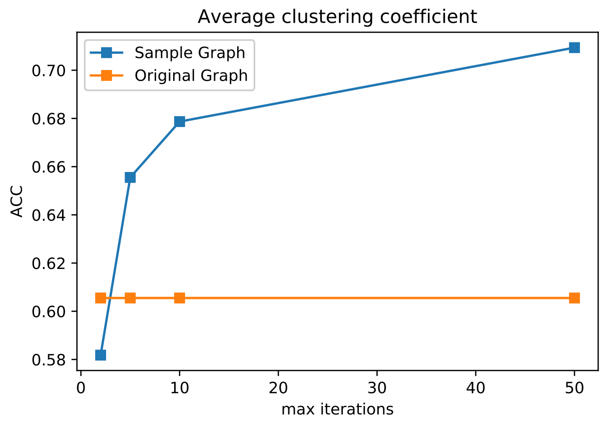


Figure 4. Average Clustering Coefficient, as maxIter differs, with alpha=5

from each one of the clusters. The sample graph returned by the algorithm, therefore, is a set of connected components, in which the nodes are strongly connected, while between the communities there are few or not at all links. This happens because the communities are quite clearly defined and compact, so the random walk on each one doesn't usually produce vertices with edges to other communities.

The second observation, which was also the purpose of modifying the maxIter parameter, is that there seem to be fewer sampled communities as the value of maxIter parameter increases. This has already been explained above and is due to the operation of the LPA algorithm, in which the more iterations we perform the more sub-clusters the algorithm merges and therefore results in slightly larger but fewer communities.

Figure 4 shows the effect of the value of maxIter, and therefore the number of created communities on the computed average clustering coefficient of the total sampled graph.

4.2 Experiments on parameter alpha

As discussed in section 3.1, the percentage of nodes to be obtained from each cluster is customly defined, and depends on two parameters: i. the number of nodes in the cluster, and ii. the average clustering coefficient of the cluster. Thus, the random walk to be conducted in each one of the clusters has a customized maximum length.

Through our experiments, we confirmed what was expected from the way this relationship is mathematically defined, and that is, the higher the alpha value, the lower the percentage of nodes we want to keep from each community, since the length of the random-walk decreases. For a fixed maxIter=5 and alpha=1 and alpha=5 we get the graphs of figure 5.

Figure 6 shows the differentiation of the clustering coefficient, based on different combinations of alpha and maxIter values. We can observe that the clustering coefficient that is

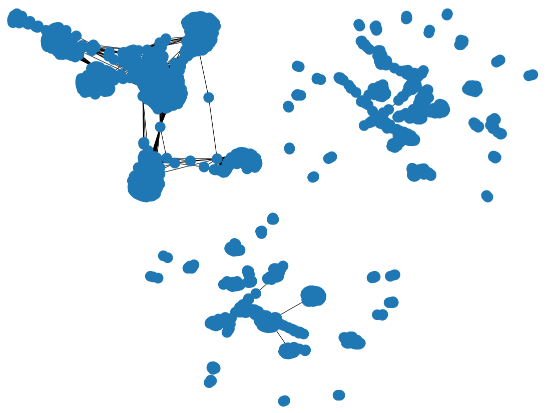


Figure 5. The original graph and its sampled versions, changing alpha. Top left: original, bottom: alpha=1, top right: alpha=5

closer to the ACC of the graph is for values alpha = 3 and maxIter = 2, although in general, the estimate of the clustering coefficient is not far from the real value, of the original graph.

4.3 Node Degree Distribution

In this section we will present you two node degree distribution histograms, both for the initial graph and for the sampled one, in 7. NDD histograms in general, depict the probability of a randomly picked node to have degree equal to k, for all the possible degrees in the graph. In most of the real-world graphs, the distribution that node degrees follow is the power-law distribution.

That can also be confirmed in this graph, both in the case of the original and the one exported by the algorithm. One can observe that while the power-law distribution is present in both charts, the range of node degrees, represented by

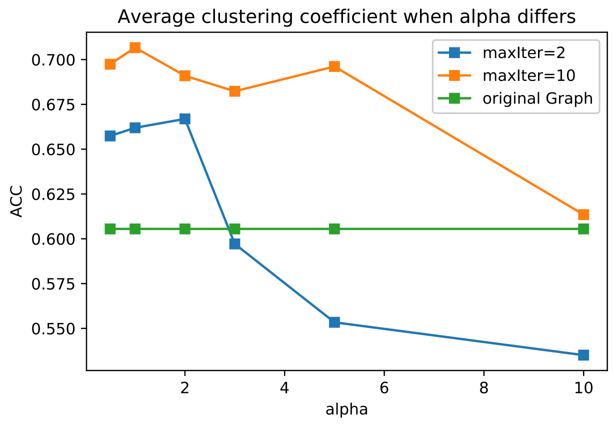


Figure 6. Average Clustering Coefficient when alpha differs from 0.5 to 10, with maxIter=2 and maxIter=10

the x-axis, changes. This is the expected behavior since we take a sample of the initial nodes and each node has fewer neighbors.

4.4 Detailed Results

For reasons of completeness, we have included in this report three detailed and descriptive tables, in which the results of the experiments performed are presented in detail, on this specific dataset. Tables 2, 3, and 4, which are located in appendix, show how some of the most known graph properties are affected with the changes in the parameter values. These metrics are **average clustering coefficient**, **average betweenness**, **average degree**, **average closeness centrality** and **transitivity**.

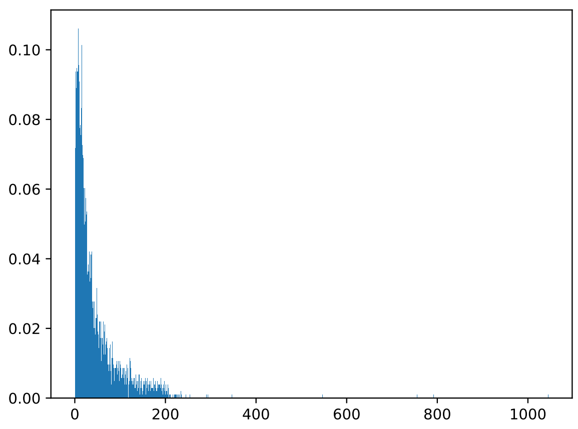
The values of the original graph regarding these properties are shown in table 1:

Average Clustering Coefficient	0.6055
Average Degree	43.691
Average Betweenness Centrality	0.0006
Transitivity	0.5191
Closeness Centrality	0.2761

Table 1. Original graph's properties

5 Conclusion and Future Work

Social networks are growing day by day. Everyday more and more people are using them and new accounts are created. Sampling is necessary for social network analysis in order to extract useful information. A good and representative sample keep all the properties of the initial graph and that's why sampling may cost a lot in resources and sometimes is not even possible. Clustering the available data and sample a part from each cluster will significant reduce time and memory consumption of the sampling procedure.



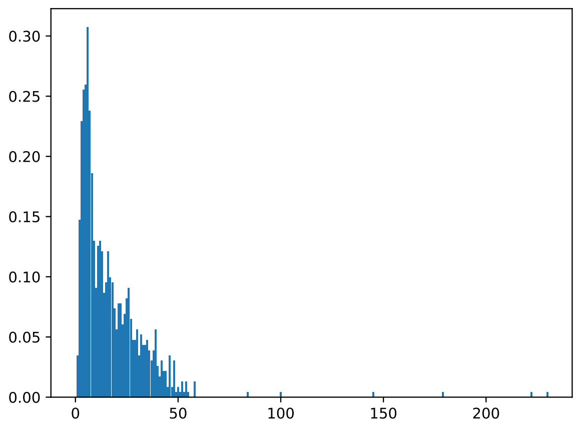


Figure 7. The original graph histogram and its sampled version. Top: original, bottom: alpha=5, maxIter=2.

The values in table 1 represent the property values from the initial graph, and are used as a comparison measure for the algorithm's results.

As we mentioned before, as the maxIter increases, we get fewer communities. This is clear and one can notice the number of communities in table 2. In table 2, we present the results for changing the maxIter parameter, alpha=5. For maxIter = 2 we have 263 communities, for maxIter = 5 we have 94 communities, for 10 we have 70 and for 50 we have 66 communities. The maximum average degree is 14.31903 and the maximum clustering coefficient is 0.709375 for maxIter = 10 and maxIter=50, respectively. For maximum iterations number greater or equal to 5, the average clustering coefficient is higher that initial's graph average clustering coefficient.

The higher the alpha value, the lower the percentage of nodes we want to keep from each community. In table 3, we present the results for changing the alpha parameter while maxIter is kept constant to 2. The maximum average degree is 16.71916 for alpha=0.5. The clustering coefficient closest to the real one is obtain for an alpha value equal to 3, while for alpha=2 we get the maximum average clustering coefficient equal to 0.666904, which is higher than the initial graph's average clustering coefficient. As we can notice, average clustering coefficient follows a downtrend for alpha greater or equal to 3. Despite that the maximum transitivity is 0.613896 for alpha=5, the transitivity with the minimum distance from the one of the original graph is given for alpha=1.

In table 4, we present the results for changing the alpha parameter when maxIter=10. The maximum average clustering coefficient is 0.706658 for alpha=1 and the minimum average clustering coefficient is 0.61348 for alpha=10. The plot in figure 6 gives a representation of these values. The maximum average degree is 19.890977 for alpha=1, while the maximum transitivity is 0.58435 for alpha=10 and the maximum average closeness centrality is 0.07759 when alpha equals to 5. For maxIter=10, average clustering coefficient is higher that the initial graph's average clustering coefficient for any value of parameter alpha.

As future work, it will be very interesting to compare the already existing sampling algorithms' performance with our method in regards to the results, but also under a cost constraint. Sampling methods such as MHRW and RWRW it is proven that reduce the bias between high-degree nodes and perform really well.

Another question is how good our method could perform with an even bigger and more complex dataset, i.e. how good scales. Would our method significantly reduce the time consumption in comparison to a non-parallel technique for sampling?

Our method returned a sample graph which is a set of connected components, in which the nodes are strongly connected, but with few or no inter-community links. The next step could be to modify our algorithm in order to return a connected graph instead of a set of strongly connected components. This modification may result in better-sampled graphs, more representative and more useful for further analysis, as a disconnected graph can be difficult in handling.

References

[1] Krista J Gile and Mark S Handcock. 2010. 7. Respondent-driven sampling: An assessment of current methodology. *Sociological methodology* 40, 1 (2010), 285–327.

[2] Minas Gjoka, Maciej Kurant, Carter T Butts, and Athina Markopoulou. 2010. Walking in facebook: A case study of unbiased sampling of osns. In *2010 Proceedings IEEE Infocom*. Ieee, 1–9.

[3] Aditya Grover and Jure Leskovec. 2016. node2vec: Scalable Feature Learning for Networks. *KDD : proceedings. International Conference on Knowledge Discovery & Data Mining* 2016, 855–864. https://doi.org/10.1145/2939672.2939754

[4] Andrew Lumsdaine, Douglas Gregor, Bruce Hendrickson, and Jonathan Berry. 2007. Challenges in parallel graph processing. *Parallel Processing Letters* 17, 01 (2007), 5–20.

[5] Zhuangkun Wei, Bin Li, and Weisi Guo. 2019. Optimal sampling for dynamic complex networks with graph-bandlimited initialization. *IEEE Access* 7 (2019), 150294–150305.

[6] Xin Xu, Chul-Ho Lee, et al. 2017. Challenging the limits: Sampling online social networks with cost constraints. In *IEEE INFOCOM 2017-IEEE Conference on Computer Communications*. IEEE, 1–9.

[7] Junpeng Zhu, Hui Li, Mei Chen, Zhenyu Dai, and Ming Zhu. 2018. Enhancing Stratified Graph Sampling Algorithms Based on Approximate Degree Distribution. In *Computer Science On-line Conference*. Springer, 197–207.

A Tables

MaxIter	avg cc	avg degree	avg betweenness centr	transitivity	avg closeness centr	#communities
2	0.5818	9.5443	0.0011529	0.497	0.0794	263
5	0.65553	10.6978	0.0004227	0.55856	0.056866	94
10	0.6787	14.31903	0.001018	0.50256	0.1531405	70
50	0.709375	14.21774	0.001716	0.4616509	0.1913427	66

Table 2. The metrics for changing the maxIter parameter, alpha=5.

Alpha	avg cc	avg degree	avg betweenness centr	transitivity	avg closeness centr	communities
0.5	0.657388	16.71916	0.000625	0.59729	0.07851	263
1	0.661937	16.67014	0.000633	0.52715	0.12886	263
2	0.666904	13.89769	0.001398	0.48168	0.18438	263
3	0.597138	11.32041	0.001244	0.5862	0.05598	263
5	0.553406	9.61104	0.002577	0.613896	0.06434	263
10	0.535084	7.11963	0.002784	0.45598	0.06091	263

Table 3. The metrics for changing the alpha parameter, maxIter=2.

Alpha	avg cc	avg degree	avg betweenness centr	transitivity	avg closeness centr	communities
0.5	0.697408	23.3982	0.00116	0.551662	0.19677	70
1	0.706658	19.890977	0.00127	0.531629	0.183311	70
2	0.690978	19.4316	0.00142	0.517244	0.18231	70
3	0.68231	13.09840	0.00073	0.59386	0.06989	70
5	0.69614	11.52137	0.00056	0.56983	0.07759	70
10	0.61348	7.71163	0.00106	0.58435	0.04699	70

Table 4. The metrics for changing the alpha parameter, maxIter=10.