TRIANGLES INTO THE WILD: A COUNTING SAFARI IN THE JUNGLE OF LARGE GRAPHS

Chatzichristodoulou Zoi
Dept. of Informatics
Thessaloniki, Greece
zchatzi@csd.auth.gr

Sofia Vei
Dept. of Informatics
Thessaloniki, Greece
sofiavei@csd.auth.gr

Maria Liatsikou
Dept of Informatics
Thessaloniki, Greece
liatsikou@csd.auth.gr

ABSTRACT

This paper implements several exact and approximate triangle counting algorithms from the literature using Python and explores their scalability properties on different real-world data sets. The implemented exact algorithms include the brute force algorithm, the node iterator algorithm, and the compact forward algorithm, while the DOULION algorithm is used for approximate triangle counting. Additionally, the streaming algorithm TRIEST is explored for triangle counting in dynamic graphs. Our experiments evaluate the performance of each algorithm on different data sets and compare their efficiency in terms of time and space complexity. The results of our study provide a comprehensive evaluation of each algorithm and its scalability properties, which will be valuable for researchers and practitioners who work with large graphs and are interested in efficient triangle counting algorithms.

Keywords: Triangle counting, exact algorithms, approximate algorithms, streaming algorithms, scalability, real-world data sets, Python, brute force algorithm, node iterator algorithm, compact forward algorithm, DOULION algorithm, TRIEST algorithm, time complexity, space complexity.

0 INTRODUCTION

Counting triangles in a graph is a fundamental problem in graph theory and has numerous applications in various fields such as social network analysis, biological network analysis, and web analytics. While exact triangle counting algorithms have been well-studied in the literature, their scalability becomes a concern when dealing with large graphs. Therefore, there has been a growing interest in developing efficient approximate and streaming algorithms for triangle counting.

In this work, we implement several exact and approximate triangle counting algorithms from the literature using Python and explore their scalability properties on different real-world data sets with varying sizes and complexities. Specifically, we implement the brute force algorithm, the node iterator algorithm the compact forward algorithm for exact triangle counting, and the Doulion algorithm for approximate triangle counting. Additionally, we explore the streaming algorithm TRIEST for triangle counting in dynamic graphs.

We evaluate the performance of each algorithm on the different data sets and compare their efficiency in terms of time and space complexity. Our goal is to provide a comprehensive evaluation of each algorithm and its scalability properties. The results of our experiments will be valuable for researchers and practitioners who work with large graphs and are interested in efficient triangle-counting algorithms.

1 PROBLEM DESCRIPTION

Triangle counting in graphs is an important problem in graph theory and network analysis, as it provides insights into the underlying structural properties of the graph. In particular, triangle counting can be used to measure the level of clustering or the degree to which nodes in a graph tend to form groups or clusters. This can be useful in a variety of applications, such as social network analysis, recommendation systems, and community detection.

One of the main challenges in triangle counting in big graphs is scalability. As the size of the graph increases, the number of triangles that need to be counted also increases, making it computationally infeasible to use traditional methods such as brute-force counting. This has led to the development of a variety of approximate and randomized algorithms for triangle counting, such as sampling-based methods and fixed-parameter algorithms (Steorts, 2015).

Another challenge in triangle counting in big graphs is dealing with the sparsity of the graph. Many large graphs, such as online social networks, are highly sparse, with a large number of nodes and few edges. This makes it difficult to accurately count triangles, as many potential triangles may not exist in the graph (Leskovec et al., 2010). To address this issue, researchers have proposed a variety of methods for dealing with sparse graphs, such as using graph sparsifiers (Spielman & Srivastava, 2011) and random walk-based methods (Riondato & Kornaropoulos, 2016).

In addition, big graphs are often dynamic and evolve. This makes it challenging to accurately count triangles, as the graph structure is constantly changing. To address this issue, researchers have proposed a variety of methods for counting triangles in dynamic graphs, such as using sliding windows (Leskovec et al., 2005) and online algorithms (Pagh et al., 2013).

In conclusion, triangle counting in graphs is an important problem in graph theory and network analysis, as it provides insights into the underlying structural properties of the graph. The main challenges in triangle counting in big graphs are scalability, sparsity, and dynamics of the graph. A variety of approximate and randomized algorithms have been proposed to address these challenges.

2 DATA

We apply the algorithms to three SNAP datasets (Leskovec & Krevl, 2014). More precisely, we choose two collaboration networks and one spatial network:

- Arxiv GR-QC (General Relativity and Quantum Cosmology) is an undirected collaboration network and it regards scientific collaborations between authors of papers submitted to the category of General Relativity and Quantum Cosmology of the e-print Arxiv[1]. Two authors are connected with a link if they are co-authors of a paper.
- Arxiv COND-MAT (Condensed Matter Physics) is another undirected collaboration network, where nodes represent authors and links represent co-authorship of papers submitted to Condense Matter category.
- A road network of California, where nodes represent intersections and undirected edges represent the roads that connect these intersections.

Both collaboration networks refer to the period from January 1993 to April 2003. The structural characteristics of the networks are summarized in Table 1. All of the networks have similar densities, but they vary a lot in size and this gives us the chance to examine the scalability of the applied algorithms.

Table 1: Structural characteristics of the networks

Name	**Nodes**	**Edges**	**Triangles**	**Triangle density**
GR-QC	5.242	14.496	48.260	9,98
COND-MAT	23.133	93.497	173.361	5,56
Road Network	1.965.206	2.766.607	120.676	0,13

As a preprocessing step, we remove all self-loops that may exist in the networks.

3 BASELINE ALGORITHMS

The three basic algorithms used are Brute-Force, Node-Iterator and Compact-Forward. Each of the three is an exact algorithm, and therefore our results in this case are accurate to all extent, thus, our concern is focused on the time complexity of each algorithm and afterwards, the comparison results they give through the sparsification of Doulion based on accuracy and time, respectively.

The Brute-Force algorithm for triangle counting finds all triplets, and if all edges appear, then it considers a detected triangle.

Thus, Brute-Force is the slowest of all algorithms detecting the triangles, nevertheless, having accurate results in all cases, since it is considered an exact algorithm. Specifically, it takes 4.320, 84.11 for the first two datasets respectively, and we approximately calculate that it would take 607015620s (almost 19 years!) for the largest dataset, which leads us to the conclusion that it is definitely not a convenient algorithm even for smaller datasets, due to its unscalability deriving from its cubic complexity which is obviously mirrored in our results.

The next algorithm, Node-Iterator, works by iterating through all nodes in the graph while checking if they form a triangle with their neighbors.

On the first dataset, Node-Iterator has, actually, good results on running time, since it takes 0.261s. For the second dataset, as the size is almost quintupled, the running time achieves 3.501s. Finally, for the roadmap dataset, Node-Iterator is even slower, taking about 7.111s to perform,

[1] https://arxiv.org/

which is justified due to the size of each network, reasonably confirming its complexity of $O(n \cdot d_{max}^2)$.

The last exact algorithm used is Compact-Forward. It is an extension of Forward algorithm, that reduces its space complexity. A counter variable is initialized. It sorts the vertices of the graph based on the degrees and for each vertex of the graph it considers its neighbours that have a higher degree, which are potential second vertices of triangles with the current vertex as one of the three vertices. For each potential second vertex, it checks if it has any common neighbours and increments the counter variable accordingly. This process is followed for all vertices of the graph.

Once again, the exact number of triangles for each dataset is achieved, as mentioned in *Table 1* above. For the first, smallest, dataset the runtime is 0.327s. For the second dataset the runtime achieved is 2.340s, less than the runtime Node-Iterator achieves, which is justified due to the smallest complexity Compact-Forward achieves, almost $O(m^{\frac{3}{2}})$. For the road network, Compact-Forward needs 41.848s to perform. The delay for this dataset when compared to the results of Node-Iterator could be due to the small clustering coefficient of the road network (0.046, while the second dataset holds a clustering coefficient of 0.663, which is much larger), since while checking the way each of the two algorithms work, smaller clustering coefficient means smaller possibility of checking the neighbours, and thus, Node-Iterator needs less time to check for triangles in the graph. Additionally, we would suggest trying different methods for sorting the nodes during the process of Compact-Forward, since this part of the algorithm takes up a large part of the runtime, and we would suggest that the consumption could be smaller.

Concluding for the baseline algorithms, Node-Iterator and Compact-Forward perform best, especially for the largest datasets. However, we proceed to using Doulion for the sparsification of our networks, to achieve even better results concerning time complexity, having the goal of even better results aiming at speedups and consequently results based on approximations.

4 DOULION

The Doulion algorithm (Tsourakakis et al., 2009) is an approximating algorithm, which is proposed for cases where exact triangle counting algorithms are not applicable due to the size of the graph. It can be used both in the case that the graph fits in the main memory and if its size exceeds the memory. For each edge of the graph, the algorithm tosses a coin and the edge is kept with probability *p* and is deleted with probability *1-p*. If the edge is kept, it is reweighted with a weight equal to *1/p*. In other words, Doulion specifies the graph. After the sparsification step, it can be combined with any triangle counting method, like NodeIterator or Compact Forward algorithms, which outputs the number of the triangles in the resulting graph *G'*. Finally, each triangle of *G'* counts as $1/p^3$ triangles. The expected number of triangles that Doulion outputs are equal to the actual number. The variance is given by the equation:

$$Var(X) = \frac{\Delta(p^3 - p^6) + 2k(p^5 - p^6)}{p^6}$$

where Δ is the total number of triangles of G and k is the number of pairs of triangles that are not edge-disjoint. Using Chebysev's inequality it is proved that

$$Pr(|X - \Delta| \geq \varepsilon\Delta \leq \frac{(p^3 - p^6)}{p^6\varepsilon^2\Delta} + 2k\frac{(p^5 - p^6)}{p^6\varepsilon^2\Delta^2}$$

As we can see, the probability that the algorithm's output approaches the actual number Δ increases as Δ gets larger and as p gets closer to 1.

In this work, we apply the Doulion algorithm both with NodeIterator and Compact Forward algorithms. We run the algorithms ten times in order to calculate the mean values of accuracy and related error and their standard deviation. We try five different values of p: 0.1, 0.3, 0.5, 0.7, and 0.9 to examine the performance of the algorithm in different levels of sparsification. Tables 2-4 summarize the results (we include only the values of the standard deviation of accuracy since the corresponding values of related error are very similar).

Table 2: Results for Doulion algorithm for GR-QC dataset

GR-QC	**p=0.1**	**p=0.3**	**p=0.5**	**p=0.7**	**p=0.9**
Accuracy	0.87	0.94	0.96	0.98	0.99
Rel. error	0.131	0.056	0.004	0.018	0.010
Std	0.185	0.048	0.020	0.013	0.007

Table 3: Results for Doulion algorithm for COND-MAT dataset

COND-MAT	**p=0.1**	**p=0.3**	**p=0.5**	**p=0.7**	**p=0.9**
Accuracy	0.94	0.97	0.99	0.99	0.99
Rel. error	0.065	0.026	0.006	0.006	0.006
Std	0.059	0.017	0.090	0.002	0.004

Table 4: Results for Doulion algorithm for Road Network dataset

Road Network	**p=0.1**	**p=0.3**	**p=0.5**	**p=0.7**	**p=0.9**
Accuracy	0.95	0.99	0.99	0.998	0.998
Rel. error	0.053	0.011	0.007	0.002	0.002
Std	0.071	0.013	0.006	0.002	0.001

It is obvious that in most cases the accuracy is higher than 0.95, while the variance is low. We also observe that in the road network, which is the largest dataset, the accuracy value remains very high, while the variance remains low, even if we only keep 10% of the edges.

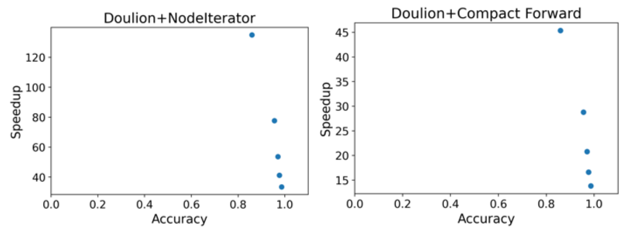

Figure 1: Results for Doulion algorithm for GR-QC dataset

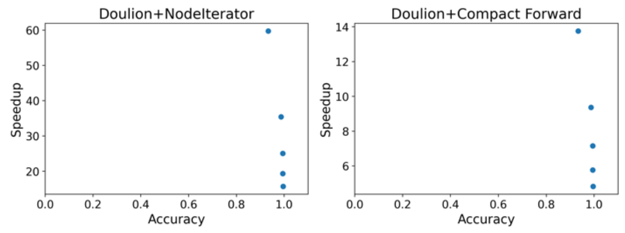

Figure 2: Results for Doulion algorithm for COND-MAT dataset

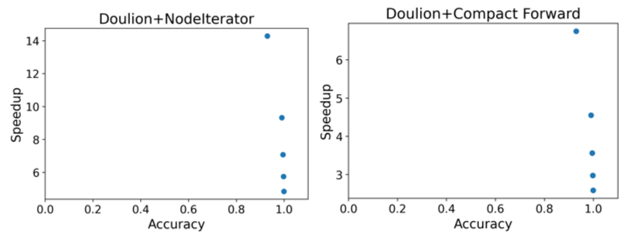

Figure 3: Results for Doulion algorithm for Road Network dataset

Figures 1-3 show the speedups in relation to accuracy for Doulion combined with NodeIterator and Compact Forward algorithms for the three datasets for the different values of *p*. The highest speedup is, as expected, achieved for p=0.1, which is, in all cases, significantly higher than those for larger values of p. We also observe that the speedup is higher as the graph gets smaller, and it is consistently higher when Doulion is combined with NodeIterator than with the Compact Forward algorithm. We discover that in the case of large road networks, which are characterized by low triangle density, the speedup is not so high as in other kinds of large networks.

5 STREAMING APPROACH

Streaming algorithms are becoming increasingly important for analyzing large-scale network data due to their low memory requirements and ability to process data in real time. One application of streaming algorithms is in the counting of triangles in a graph, which is a fundamental task in network analysis. In this work, we explore the importance of streaming algorithms for triangle counting and their relevance in today's data-driven world. Counting the number of triangles in a graph can help us understand the structure of a network and identify important nodes or communities. However, as the size of the network grows, traditional algorithms for triangle counting become computationally expensive and memory-intensive. In contrast, streaming algorithms offer a solution that is both efficient and scalable. The importance of streaming algorithms for triangle counting lies in their ability to handle large-scale network data, which is becoming increasingly common in today's data-driven world. With the explosive growth of social media, e-commerce, and other online services, the need for efficient and scalable algorithms for network analysis has become more pressing than ever. Streaming algorithms offer a solution that can process data in real time and with limited memory resources.

Streaming algorithms work by processing data in a single pass, with limited memory available for storing the data. The TRIEST algorithm is a popular example of a streaming algorithm for triangle counting, which maintains a reservoir of a fixed size that is used to sample edges from the graph. By sampling edges uniformly at random from the reservoir, the algorithm can estimate the number of triangles in the graph. The TRIEST algorithm has been shown to be effective for large-scale graphs with limited memory resources (Rossi et al., 2014).

Other algorithms for triangle counting have also been developed, such as the TGIC algorithm, which uses a combination of graph partitioning and matrix multiplication to efficiently count triangles in a distributed computing environment. The TTC algorithm is another example of a streaming algorithm that uses a probabilistic approach to estimate the number of triangles in a graph (Bhatia et al., 2019).

For the purposes of this work, we implement the improved TRIEST algorithm in Python and test it on three real-world sparse graphs with different memory reservoir sizes. To implement the TRIEST algorithm, we use the NetworkX library for graph manipulation and the random module for edge sampling. We start by initializing the reservoir with k-selected edges from the graph. We then stream the remaining edges and update the reservoir and triangle count as described in the algorithm.

The accuracy of the TRIEST (TRIestimate) algorithm for triangle counting depends on several factors, including the size of the memory reservoir, the sparsity of the graph, and the degree distribution of the nodes in the graph (Tsourakakis 2018).

The size of the memory reservoir is an important factor in determining the accuracy of the algorithm. The larger the reservoir, the more edges are sampled from the graph, which can improve the accuracy of the estimated triangle count. However, increasing the reservoir size also increases the memory requirements of the algorithm, which may be a limiting factor for some applications.

The sparsity of the graph is also important, as it affects the probability of sampling triangles. Dense graphs have a higher probability of containing triangles, which can lead to more accurate estimates with smaller memory reservoirs. In contrast, sparse graphs may require larger memory reservoirs to achieve the same level of accuracy.

The degree distribution of the nodes in the graph can also affect the accuracy of the algorithm. In graphs with a power-law degree distribution, where a few nodes have a very large degree and most nodes have a small degree, the accuracy of the algorithm may be affected as the probability of sampling triangles from these high-degree nodes is higher. This can lead to an overestimation of the number of triangles in the graph.

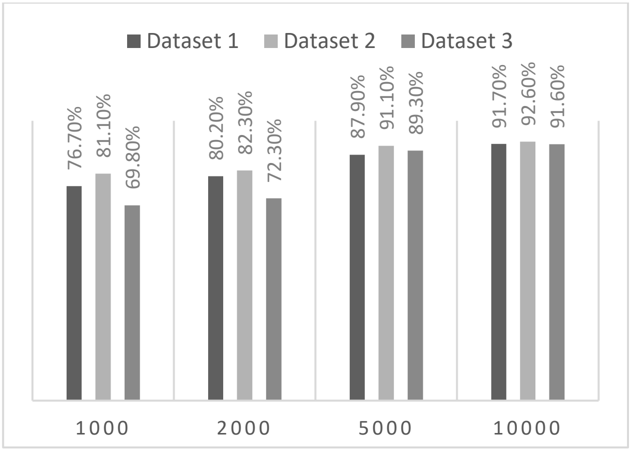

Figure 4 Results from TRIEST Application

In this work, we aim to test the scalability of the TRIEST algorithm so the test that we perform involves testing on the three real-world sparse graphs described in chapter 2 which have drastically different sizes yet similar density.

Furthermore, we run the algorithm for different memory reservoir sizes (M = 1000, 2000, 5000, and 5000 of the total edges) and record the estimated triangle count. As an evaluation metric, we use accuracy calculated as the number of estimated triangles divided by the actual number of triangles. Additionally, to smooth out the effect of randomness, results were averaged over ten independent total trials. Figure 4 holds the final results of the test.

We observe significant improvement as the memory reservoir increases. The best overall result is found in the second dataset with minimum accuracy at 81.1% with even 1000 edges memory reservoir which is less than 9% of the whole size of the graph. Finally, the scalability capability is more efficiently captured by the third dataset that scores 91,6% accuracy with a 10.000 memory reservoir which corresponds to less than 1% of the whole size of the graph.

6 CONCLUSIONS

Firstly, we apply three baseline algorithms, Brute-Force, Node-Iterator, and Compact-Forward, checking the runtimes of all three, while concluding that especially Brute-Force is extremely time-consuming, especially for the largest, road network. We note here that the extremely small clustering coefficient of the last dataset leads to Compact-Forward demanding more time than Node-Iterator – which, in general, is much faster - that needs fewer checks of the neighbors, while additionally, as future work we would suggest trying different sorting methods for Compact-Forward as this part takes up a large amount of the process running time and we would suggest that it could reduce even more time complexity.

We then apply the Doulion algorithm both with NodeIterator and Compact Forward algorithms for different values of p. The values of accuracy are high (with low variance across our runs), even in the case we keep only 10% of the edges. Regarding the speedups achieved by Doulion, the highest speedup is achieved for p=0.1, which is, in all cases, much higher than those for larger values of p. It is consistently higher when Doulion is combined with NodeIterator than with Compact Forward and the values achieved are higher as the graph gets smaller.

The TRIEST algorithm for estimating the number of triangles in a graph depends on several factors, including the size of the graph, the sampling rate, and the size of the memory reservoir used by the algorithm. The relationship between these factors and accuracy can be complex and dependent on other factors, but both research studies and applications have shown that the algorithm can be effective and accurate in a variety of real-world settings with the appropriate tuning of these parameters. Overall, the TRIEST algorithm offers a powerful and efficient approach to triangle counting in large, dynamic graphs, and can be a valuable tool for data scientists and researchers working with network data.

7 CODE AND DATA AVAILABILITY

The code, the data, and the evaluation results used in the proposed solution are available on GitHub in:

https://github.com/zoichatzi/graphTriangleCounting

8 REFERENCES

Leskovec, J., Lang, K. J., Dasgupta, A., & Mahoney, M. W. (2010). Empirical comparison of algorithms for network community detection. In Proceedings of the 19th international conference on World wide web (pp. 631-640). ACM.

Leskovec, J., Kleinberg, J., & Faloutsos, C. (2005). Graphs over time: densification laws, shrinking diameters, and possible explanations. In Proceedings of the eleventh ACM SIGKDD international conference on Knowledge discovery in data mining (pp. 177-187). ACM.

Pagh, R., Riondato, M., & Silvestri, F. (2013). Dynamic triangle counting in large graphs. In Proceedings of the 2013 ACM SIGMOD international conference on Management of data (pp. 823-834). ACM.

Riondato, M., & Kornaropoulos, E. (2016). Fast triangle counting in large graphs through a combination of the sandwich method and pruning. In Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining (pp. 929-938). ACM.

Spielman, D. A., & Srivastava, N. (2011). Graph sparsification by effective resistances. In Proceedings of the 43rd ACM Symposium on Theory of Computing (pp. 513-522).

Leskovec, J., & Krevl, A. (2014). SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data.

Tsourakakis, C. E., Kang, U., Miller, G. L., & Faloutsos, C. (2009, June). Doulion: counting triangles in massive graphs with a coin. In Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining (pp. 837-846).

Rossi, R. A., Ahmed, N. K., & Neville, J. (2014). Triest: Counting local and global triangles in fully-dynamic streams with fixed memory size. In Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining (pp. 1356-1365).

Bhatia, S., Chitnis, R., & Gupta, A. (2019). Large-Scale Triangle Counting: A Survey. Journal of Combinatorial Optimization, 38(2), 464-502.

Tsourakakis, C. E., Kapralov, M., & Karypis, G. (2018). Sublinear algorithms for (Δ+ 1)-coloring and triangle counting on sparse graphs. Journal of the ACM (JACM), 65(6), 1-37.