Intersection Representation of Big Data Networks and Triangle Counting

Wali Mohammad Abdullah
Mathematics and Computer Science
University of Lethbridge
Lethbridge, Alberta, Canada
w.abdullah@uleth.ca

David Awosoga
Mathematics and Computer Science
University of Lethbridge
Lethbridge, Alberta, Canada
david.awosoga@uleth.ca

Shahadat Hossain
Mathematics and Computer Science
University of Lethbridge
Lethbridge, Alberta, Canada
shahadat.hossain@uleth.ca

***Abstract*—Triangles are an essential part of network analysis, representing metrics such as transitivity ratio and clustering coefficient Because of its diverse applications, enumeration and counting of triangles in large networks has been extensively studied, and continues to draw much interest from many different fields. This has only increased with the introduction of approximate counting, parallel and distributed implementations, and restricted and streaming data access scenarios. We propose a compact and efficient representation of network data based on the *intersection* of edge labels, and use sparse matrix data structures for its computer implementation. We then present a scalable algorithm that uses this structure to count triangles. On a set of large (the largest with more that 3.6 billion edges) real-world and synthetic networks, our algorithm performs significantly better than the reference implementation miniTri [1].**

***Index Terms*—Intersection matrix, Triangle count, Forward degree cumulative, Forward neighbors, Sparse graph**

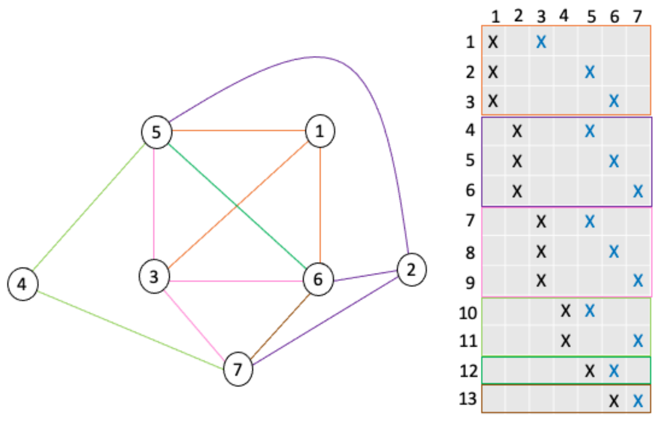
I. Introduction

The presence of triangles in network data has led to the creation of many metrics to aid in the analysis of graph characteristics and network evolution over time [2]. Efficiently representing network data is essential to improving analysis capabilities, algorithm performance, and data visualization potential [3]. Large real-life networks are typically sparse in nature, which does not bode well for the traditionally used adjacency matrix representation that requires much more memory than necessary and does not scale well. In this paper, we propose an "intersection" representation [4] of network data based on sparse matrix data structures [5]. After describing the triangle counting algorithm in terms of linear algebra kernel operations, we demonstrate numerically that our implementation is highly effective. We present the comparative running times with reference implementation miniTri [1] and show that our method scales very well on massive network data.

II. Intersection Matrix Representation

Let $G = (V, E)$ be an undirected and connected graph without self-loops or multiple edges between a pair of vertices. Let the vertices in V be labelled $1, 2, \ldots, |V| = n$. Using the labels on the vertices, a unique label can be assigned to each edge $e_k = \{v_i, v_j\}, i < j, k = 1, 2, \ldots, |E| = m$.

The *intersection representation* of graph G is a matrix $X \in \{0,1\}^{m \times n}$ in which for each column j of X there is a vertex $v_j \in V$ and $\{v_i, v_j\} \in E$ whenever there is a row k for which $X(k,i) = 1$ and $X(k,j) = 1$. The rows of X represent the edge list sorted by vertex labels. Therefore, matrix X can be viewed as an assignment to each vertex a subset of m labels such that there is an edge between vertices i and j if and only if the inner product of the columns i and j is 1. Since the input graph is unweighted, the edges are simply ordered pairs, and can be sorted in $O(m)$ time. Unlike the adjacency matrix which is unique (up to a fixed labelling of the vertices) for graph G, there can be more than one *intersection matrix* representation associated with graph G [6]. We exploit this flexibility to store a graph in a structured and space-efficient form.



	1	2	3	4	5	6	7
1	X		X				
2	X				X		
3	X					X	
4		X			X		
5		X				X	
6		X					X
7			X		X		
8			X			X	
9			X				X
10				X	X		
11				X			X
12					X	X	
13						X	X

Fig. 1. Example Input Graph and its Intersection Matrix Representation

A. Adjacency Matrix-based Triangle Counting

Many of the existing triangle counting methods use the sparse representation of adjacency matrices in their calculations. The adjacency matrix $A(G) \equiv A \in \{0,1\}^{|V| \times |V|}$ associated with graph G is defined as,

$$A(i,j) = \begin{cases} 1 & \text{if } \{v_i, v_j\} \text{ where } i \neq j \text{ is in } E \\ 0 & \text{otherwise} \end{cases}$$

2021 IEEE Big Data Camera Ready Version
To cite this paper please use the final published version:
DOI: 10.1109/BigData52589.2021.9671349

It is well known in the literature that the number of closed walks of length $k \geq 0$ are obtained in the diagonal entries of kth power A^k such that the total number of triangles in a graph $G, \Delta(G)$, is given by the trace of A^3,

$$\Delta(G) = \frac{1}{6} Tr(A^3).$$

The factor of $\frac{1}{6}$ accounts for the multiple counting of a triangle (the number of ways closed walks of length 3 can be obtained is 3!). There is a large body of literature on sparse linear algebraic triangle counting methods based on adjacency matrix representation of the data [3]. miniTri's triangle counting implementation takes the adjacency matrix A of the input graph and creates an incidence matrix B from it [1]. The enumeration and counting of the triangles occur in the overloaded matrix multiplication $C = AB$, where entries in the resultant matrix C with a value of 2 correspond to a completed triangle. This method triple counts each triangle, once for each vertex, so the final result is divided by 3 giving the total number of triangles in the graph. This is a memory intensive process, since the multiplication of two sparse matrices usually results in a dense matrix.

B. Intersection Matrix-based Triangle Counting

Graph algorithms can be effectively expressed in terms of linear algebra operations [7], and we combine this knowledge with our proposed data representation to count the triangles in a structured three-step method. For vertex i we first find its neighbors $j > i$ such that $(i, j) \in E$ by multiplying the submatrix of X consisting of rows corresponding to edges incident on i (let us call them $(i-j)-$rows) by the transpose of the vector of ones of compatible length. A value of 1 in the vector-matrix product indicates that the corresponding vertex j is a neighbor of vertex i.

Next, we multiply the submatrix of X consisting of columns j identified in the previous step and the rows below the $(i-j)-$rows by a vector of ones of compatible length. A value of 2 in the matrix-vector product indicates a triangle of the form (i, j, j') where j and j' are neighbors of vertex i with $j < j'$. Let k be the row index in matrix X for which the matrix-vector product contains a 2. Then it must be that $X(k, j) = 1$ and $X(k, j') = 1$. Since each row of X contains exactly 2 nonzero entries that are 1, it follows that $(j, j') \in E$.

The number of triangles in the graph is given by the sum of the number of triangles associated with each vertex as described. Since the edges are represented in sorted order in our algorithm, unlike many other triangle counting methods [1], each triangle is counted exactly once. Figure 1 displays a graph with 7 vertices and 13 edges, along with its intersection matrix X. The triangles of the form $(1, j, j')$ where $j, j' \in \{3, 5, 6\}$ are obtained from the product $X(7:13, [3\ 5\ 6]) * \mathbf{1}$ where $\mathbf{1}$ denotes the vector of ones. The product has a 2 at locations corresponding to rows $7, 8, 12$ of X and the associated triangles are $(1, 3, 5), (1, 3, 6)$, and $(1, 5, 6)$. Thus, there are three triangles incident on vertex 1, and it can be easily verified that the graph contains a sum of 7 triangles across all of the vertices.

.

C. Data Structure

In our preliminary implementation, we use two additional arrays to store some useful information that can be computed after we sort the edges. `FDC` is an array of size n, where its elements correspond to the total number of "forward neighbors" across the vertices of the graph. Forward neighbors are defined as the neighbors of a vertex that have a higher label than the vertex of interest. With the vertices of the graph labelled, finding the forward degree of a vertex j can be calculated as `fd(j) = FDC[j+1] - FDC[j]`. `FN` is an array of size m that stores *which* vertices are the forward neighbors of a vertex j. Using `FN` we can find these forward neighbors of j as `fn(j) = FN[k]`, where k ranges from `FDC[j]` to `FDC[j+1]-1`. The arrays `FDC` and `FN` thus save the vector-matrix products needed to find the forward neighbors. Figure 2 displays the arrays `FDC` and `FN` for the graph of Figure 1.

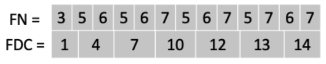


Fig. 2. FN and FDC for the Example Graph

III. Numerical Results

In this section, we provide results of numerical experiments of selected test instances. The first set contains real-world social network instances from the Stanford Network Analysis Project (SNAP), obtained from the Graph Challenge website [8]. SNAP is a collection of more than 50 large network datasets containing a large number of nodes and edges, including social networks, web graphs, road networks, internet networks, citation networks, collaboration networks, and communication networks [9]. The first set of experiments were performed using a Dell Precision T1700 MT PC with a 4th Gen Intel Core I7-4770 Processor (Quad Core HT, with 3.4GHz Turbo and 8GB RAM), running Centos Linux v7.9. The implementation language was C++ and the code was compiled using $-O3$ optimization flag with a g++ version 4.4.7 compiler.

TABLE I
Comparing Our Intersection Algorithm with miniTri on Large Real World Networks

Graph	$\lvert V\rvert$	$\lvert E\rvert$	$\Delta(G)$	miniTri	Intersection	Speedup
Loc-gowalla	196591	950327	2273138	143.7	0.234	615
roadNet-PA	1090920	1541898	67150	1.249	0.056	23
roadNet-TX	1393383	1921660	82869	1.537	0.071	22
flickrEdges	105938	2316948	107987357	876.4	1.544	568
amazon0312	400727	2349869	3686467	22.7	0.199	115
amazon0505	410236	3356824	3951063	24.73	0.212	117
amazon0601	403394	3387388	3986507	24.28	0.214	114
roadNet-CA	1965206	5533214	120676	2.224	0.102	22
Cit-Patents	3774768	33037894	7515023	146.3	2.043	72
com-friendster[a]	65608366	3612134270	1175498452	N/A	659.37	N/A

[a]The com-friendster instance was run on a Compute Canada machine.

In Table I the CPU running times of the miniTri and Intersection implementation on the 10 largest real-world test instances are compared. Times are computed in seconds, and "N/A" denotes an instance where miniTri failed to compute. The speedup of our intersection method over miniTri ranges from a modest $22\times$ to an impressive $568\times$.

TABLE II
COMPARING OUR INTERSECTION ALGORITHM WITH MINITRI ON LARGE SYNTHETIC NETWORKS

Graph	$\lvert V\rvert$	$\lvert E\rvert$	$\Delta(G)$	miniTri	Intersection
graph500-scale18-ef16	262144	4194304	82287285	17440	9.357
graph500-scale19-ef16	524288	8388608	186288972	49211.8	25.21
graph500-scale20-ef16	1048576	16777216	419349784	197456	72.34
graph500-scale21-ef16	2097152	33554432	935100883	N/A	171.2
graph500-scale23-ef16	8388608	134217728	254165706	N/A	1274.78
graph500-scale24-ef16	16777216	268435456	1346226968	N/A	3386. 2

Table II compares our algorithm performance on large synthetic test instances from GraphChallenge to miniTri. Due to the sizes of the second set of instances, they were run on the large High Performance Computing system (Graham cluster) at Compute Canada. On the first 3 instances, our method is over 1800 times faster than miniTri, and the relative performance improves with increasing instance size, further demonstrating the scalability of our triangle counting algorithm.

TABLE III
TESTING OUR INTERSECTION ALGORITHM ON NETWORKS WITH BILLIONS OF TRIANGLES

Graph (bn-human-)	$\lvert V\rvert$	$\lvert E\rvert$	$\Delta(G)$	Intersection
BNU_1_0025889_sess_2	742,862	131,926,773	14550152774	191.1
BNU_1_0025873_sess_2-bg	692,397	140,102,158	16480195100	203.9
BNU_1_0025868_sess_1-bg	727,487	150,443,553	18944842260	268.6
BNU_1_0025918_sess_1	748,521	159,835,566	23287278951	321.7
Jung2015_M87112427	728,874	171,231,873	21603267930	282.8
Jung2015_M87118465	774,886	181,569,095	24857761263	324.2
Jung2015_M87104201	707,284	191,224,983	25942442887	342.2
Jung2015_M87101705	776,644	201,198,184	26497580631	364.8
Jung2015_M87125286	753,905	209,976,387	34150382906	426.9
Jung2015_M87113878	784,262	267,844,669	41727013307	538.5

Finally, Table III demonstrates our algorithm's performance on relatively dense brain networks from [10], back on the linux environment. miniTri was unable to compute results for any of the instances, so their column was omitted.

Samsi et al. [11] provide a comprehensive review of the state-of-the-art triangle counting algorithms from the 2017-2019 MIT/Amazon/IEEE Graph Challenges, analyzing and comparing the performance of different submissions by fitting a model of graph counting times, T_{tri}, as a function of the number of edges $N_e = |E|$. They then use this data to estimate the parameters N_1 (the number of edges that can be processed in one second) and β:

$$T_{tri} = (N_e/N_1)^{\beta}$$

to compare different counting implementations. Submissions with a larger N_1 and smaller β perform the best, and the top entries from 2019 had N_1 values ranging from 5×10^5 to 5×10^8, and β values ranging from $\frac{1}{2}$ to $\frac{4}{3}$. Our algorithm had $\beta = \frac{3}{4}$ and $N_1 = 1 \times 10^7$, values competitive with the top submissions.

IV. CONCLUSION

Network data is usually input as a list of edges which can be preprocessed into a representation such as an adjacency matrix or adjacency list, suitable for algorithmic processing. We have presented a simple yet flexible scheme based on intersecting edge labels, the intersection matrix, for the representation of and calculation with network data. A new linear algebra-based method exploits this intersection representation for triangle counting – a kernel operation in big data analytics. The computational results from a set of large-scale synthetic and real-world network instances clearly demonstrate that our basic implementation is efficient and scales well. The two arrays `FDC` and `FN` together constitute a compact representation of the sparsity pattern of network data, requiring only $n+m+1$ units of storage. This is incredibly useful in the exchange of network data, with the potential to allow for many additional intersection matrix-based network analytics and algorithms.

V. ACKNOWLEDGMENT

This research was supported in part by NSERC Discovery Grant (Individual), NSERC Undergraduate Student Research Award, and the AITF Graduate Student Scholarship. A part of our computations were performed on Compute Canada HPC system (http://www.computecanada.ca), and we gratefully acknowledge their support.

REFERENCES

[1] M. M. Wolf, J. W. Berry, and D. T. Stark, "A task-based linear algebra building blocks approach for scalable graph analytics," in *2015 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2015, pp. 1–6.

[2] M. Al Hasan and V. S. Dave, "Triangle counting in large networks: a review," *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, vol. 8, no. 2, p. e1226, 2018.

[3] P. Burkhardt, "Graphing trillions of triangles," *Information Visualization*, vol. 16, no. 3, pp. 157–166, 2017.

[4] E. Szpilrajn-Marczewski, "A translation of sur deux propriétés des classes d'ensembles by," *Fund. Math*, vol. 33, pp. 303–307, 1945.

[5] M. Hasan, S. Hossain, A. I. Khan, N. H. Mithila, and A. H. Suny, "DSJM: a software toolkit for direct determination of sparse Jacobian matrices," in *International Congress on Mathematical Software*. Springer, 2016, pp. 275 – 283.

[6] W. M. Abdullah, S. Hossain, and M. A. Khan, "Covering large complex networks by cliques—a sparse matrix approach," in *Recent Developments in Mathematical, Statistical and Computational Sciences*, D. M. Kilgour, H. Kunze, R. Makarov, R. Melnik, and X. Wang, Eds. Cham: Springer International Publishing, 2021, pp. 117–127.

[7] J. Kepner and J. Gilbert, *Graph algorithms in the language of linear algebra*. SIAM, 2011.

[8] S. Samsi, V. Gadepally, M. Hurley, M. Jones, E. Kao, S. Mohindra, P. Monticciolo, A. Reuther, S. Smith, W. Song, D. Staheli, and J. Kepner, "Static graph challenge: Subgraph isomorphism," http://graphchallenge.mit.edu/data-sets, IEEE HPEC, 2017, accessed: 2021-07-09.

[9] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection," http://snap.stanford.edu/data, Jun. 2014, accessed: 2019-10-02.

[10] R. A. Rossi and N. K. Ahmed, "The network data repository with interactive graph analytics and visualization," in *AAAI*, 2015. [Online]. Available: https://networkrepository.com

[11] S. Samsi, V. Gadepally, M. Hurley, M. Jones, E. Kao, S. Mohindra, P. Monticciolo, A. Reuther, S. Smith, W. Song, D. Staheli, and J. Kepner, "Graphchallenge.org triangle counting performance," IEEE HPEC, 2020.