Triangle Counting: Problem and Related Literature

Sam Silvestro, University of Texas at San Antonio

Provides an introduction to the problem of triangle counting, as well as the review of the paper "Triangle Listing in Massive Networks" by Shumo Chu and James Cheng of the University of Hong Kong. Additionally, literature related to this problem is discussed.

1. INTRODUCTION

1.1. What is the triangle counting problem?

Given a (usually undirected) graph G=(V, E), we wish to count or list all cycles of length 3. In other words, a triangle is a "complete subgraph of G" consisting of three vertices.

1.2. Why is it important?

The concept of triangles is at the heart of the definitions of many important measures for network analysis, such as the clustering coefficients (of a single vertex in the network as well as the entire network itself), transitivity, triangular connectivity, etc.

Each of these measures can be directly computed from the result of triangle listing, and have a large number of important applications. In addition, triangle listing also has a broad range of applications in other areas, such as the discovery of dense subgraphs, the computation of trusses (i.e., subgraphs of high connectivity), spam detection, the uncovering of hidden thematic relationships in the Web, etc. In all these applications, triangle listing plays an important role.

The triangle counting problem is divided into several sub-categories. There are counting algorithms which "simply" count the number of triangles present, as well as listing algorithms, in which each triangle is enumerated. Furthermore, there are approximating versus exact algorithms, in-memory versus on-disk algorithms, etc.

Finding more efficient algorithms is a high priority due to the size of graphs being analyzed, for example, those modeling large telecommunication networks, social networks, the Internet, etc. There may be millions of nodes and billions of edges in these cases.

2. THE PAPER

The paper I explored in depth was "Triangle Listing in Massive Networks" by [Chu and Cheng 2012]. In it, the authors present an exact listing algorithm. Specifically, they focus on designing an efficient algorithm for listing when the input graph is too large to be occupied in main memory and therefore must remain on disk during processing. They point out that all of the existing algorithms fall into the category of "in-memory" algorithms, many of which still require at least linear additional memory to operate in.

While a number of approximating algorithms exist which avoid the need to parse the entire graph, their shortcoming is that they only provide an estimate of the triangle count. The authors further contend that triangle counting has significantly limited scope as compared to triangle listing: "Moreover, the set of applications of triangle counting is only a small subset of that of triangle listing, as the result of triangle counting is directly obtainable from that of triangle listing."

Their approach consists of partitioning the input graph into subgraphs that are able to fit in memory, where they process the local triangle listings in memory. Furthermore, they categorize the types of triangles they encounter into three distinct classes: Type 1, Type 2, and Type 3. Type 1 triangles are those in which all three edges of the triangle are found within a single subgraph. Type 2 triangles are those in which two edges of the triangle reside in one subgraph, while the third edge is in another. Similarly, Type 3 triangles occur when each of its three edges occur in different subgraphs.

Their algorithm assumes the data stored on disk to be in an adjacency list format, in which each vertex is assigned a unique vertex ID, and the vertices are arranged in the list in ascending order of their ID's. The algorithm itself begins like many others, in that we visit each node in-turn according to its vertex ID, then perform an intersection between its own adjacency list and that of each neighbor, but only if the neighbor has a higher vertex ID. This condition prevents duplicate processing of vertices, and therefore the duplicate outputting of triangles.

The unique vertex ID is not ordered based on the degree of the vertices – as is common in other counting algorithms – but rather, is an arbitrary numbering. This saves costly pre-processing of the adjacency list necessary for some other algorithms.

The authors point out that most existing algorithms require random access to the adjacency lists of a nodes neighbors. While the adjacency list of a node is easily read sequentially from disk while reading in the graph, the adjacency list of its neighbors could be located anywhere, necessitating random access to the disk. However, other algorithms such as "forward" require an additional storage array, the total size of which is linear in terms of its input size. This would require the extra arrays be stored on disk (unless the graph were merely a fraction of memory size), which places it too in a position to require random access to the disk.

The main idea of the algorithm is to iteratively partition the graph into pieces that can fit into main memory, then perform the triangle listing on each local subgraph. It then removes from the overall graph any edges in the subgraph that no longer contribute to triangle listing. This process is repeated until the graph is empty.

To implement this, they introduce the concept of extended subgraphs, which are simply the same as each subgraph with the addition of a directed edge connecting each vertex in the subgraph to its neighbors immediately outside of the subgraph. We can then discovery and report all Type-1 and Type-2 triangles from the extended subgraph, then remove all of the triangle edges found in the subgraph from the larger graph. After each sub-graph has been processed, the graph is repartitioned and the process repeated. By re-partitioning the shrunken graph after processing its subgraphs, we will gradually convert the set of Type-3 triangles existing during the previous iteration into Type-1 and Type-2 triangles with respect to the new partition during subsequent iterations.

If the shrunken graph becomes small enough to fit into memory at any point, the algorithm will compute a partition consisting of only one subgraph, which will eliminate any remaining Type-3 triangles with respect to the old partition. This process continues until all edges in the graph have been removed.

3. RELATED WORK

This paper directly cited the work of [Schank and Wagner 2005] titled, "Finding, Counting and Listing all Triangles in Large Graphs, An Experimental Study". The authors plainly state that the purpose of this paper is to evaluated "the practicability" of triangle counting and listing in very large graphs of various degree distributions. Each algorithm is evaluated against both artificially generated graphs, as well as real-world data (for example, the road network of Germany). They also claim to provide a simple enhancement to a well-known algorithm, making it feasible for use on very large graphs.

While this paper does not seem to contribute much in the way of original algorithm development, its findings relating to real-world runtimes of popular algorithms are cited by many papers concerning the problem of triangle counting.

The authors chose to implement each algorithm in C++, running them on a 64-bit AMD Opteron-based machine. They count the number of "triangle operations" performed by each algorithm, on graphs of increasingly greater size, to essentially describe their asymptotic running times provided no preprocessing. A triangle operation is considered to be a triangle test; that is, determining whether a triangle exists by performing some test of equality.

Their findings indicate that the two known standard algorithms namely, node-iterator and edge-iterator perform the same asymptotically (as expected). They note that edge-iterator performs best on graphs in which the degree of nodes does not differ significantly from the average. In situations in which the distribution of the degree of nodes is skewed, an algorithm such as "forward" is more suitable, and performs better asymptotically.

"Efficient Algorithms for Large-Scale Local Triangle Counting" by [Becchetti et al. 2010], explores the problem of approximating local triangle counts in large graphs. An algorithm addressing such a problem will provide a count of the number of triangles incident to every node in the graph. These authors present two such approximating algorithms. The local triangle count for a given node could then be used directly to compute its clustering coefficient. (The clustering coefficient of a node is a measure of connectedness, and is defined as the ratio of the number of incident triangles to the total possible number of incident triangles.)

The first of these algorithms is referred to in the topic paper, "Triangle Listing in Massive Networks." The authors dismiss this (and similar) algorithms which operate from external storage on the basis that it is an approximate counting algorithm, rather than the superior listing-type algorithm they provide.

Both proposed algorithms rely on "well-established probabilistic techniques" to estimate the size of the intersection of two sets. In this example, given a pair of nodes, we would like to estimate the size of the intersection of their sets of neighbors. The first algorithm performs its work with the graph stored on disk as an adjacency matrix, while the other operates in memory. They further impose on themselves the restriction of sequential access to the disk. This requirement ensures their algorithms space and time complexity are compatible with the "semi-streaming" computation model.

"Streaming" algorithms come up frequently when studying triangle counting algorithms. The streaming model was defined by [Raghavan 1999] in "Computing on data streams." It presents a model for computation in which a small (i.e., constant) number of sequential passes over the input data are allowed. Additionally, polylog space is often prescribed for streaming algorithms. Relating to graphs specifically, a graph will be read in as a "graph stream" as a sequence of edges. The "semi-streaming model" was then proposed by [Feigenbaum et al. 2005] to loosen these requirements by allowing $O(n \text{ polylog } n)$ space and logarithmically many passes.

In "DOULION: Counting Triangles in Massive Graphs with a Coin" by [Tsourakakis et al. 2009], a novel framework is proposed for counting triangles in very large graphs. Their proposal relies on the use of some other counting algorithm as a black box that we "drop in" to their framework. They implemented their framework using the Node-Iterator algorithm and performed 166 experiments on both real-world and synthetic data sets, and demonstrate speedups of up to 130 times.

The general idea is quite simple: it seeks to "sparsify" the input graph, thus yielding a speedup when counted using the black box algorithm. The framework takes as input the graph, as well as a "sparsification parameter" p, where $0 < p \leq 1$. We will use a random number generator to implement a virtual coin having a biased probability of success equal to p. We then visit each edge and "flip" our virtual coin. If the flip was successful, we keep the edge; otherwise, we delete it. (In lieu of "keeping or deleting" the edge we could simply assign a weight to it instead: $\frac{1}{p}$ on success, 0 otherwise.)

Upon completion, we provide the modified graph as input to the counting algorithm of our choosing. We then take the resulting triangle count and multiply it $\frac{1}{p^3}$ to obtain the estimated number of triangles in the original graph. If we chose to use weighted edges rather than the deletion of edges, we could modify our black box algorithm to count triangles by multiplying together their edge weights. In the event that any edge weight were 0 then the product would be 0 as well. Similarly, if all three edges possess weight $\frac{1}{p}$, their product would

yield $\frac{1}{p^3}$. As mentioned, due to the random removal of edges, this framework is properly classified as an approximating algorithm; however, they claim typical accuracy rates of over 99% in practice.

4. CONCLUSIONS

The paper and related works discussed here hardly begin to attack even a seemingly simple problem such as triangle counting. With increased reliance on this problem to compute valuable statistics relating to networks of various types, further research is bound to not just continue, but increase. Moreover, the scope of such "related works" encompasses many varied disciplines, and overlaps with several other research problems such as triad census, circuit counting, parallel processing, and many others which I explored while writing this paper. This problem will undoubtedly remain the topic of further research for some time to come.

REFERENCES

Luca Becchetti, Paolo Boldi, Carlos Castillo, and Aristides Gionis. 2010. Efficient algorithms for large-scale local triangle counting. *ACM Transactions on Knowledge Discovery from Data (TKDD)* 4, 3 (2010), 13.

Shumo Chu and James Cheng. 2012. Triangle listing in massive networks. *ACM Transactions on Knowledge Discovery from Data (TKDD)* 6, 4 (2012), 17.

Joan Feigenbaum, Sampath Kannan, Andrew McGregor, Siddharth Suri, and Jian Zhang. 2005. On graph problems in a semi-streaming model. *Theoretical Computer Science* 348, 2 (2005), 207–216.

Monika R Henzinger Prabhakar Raghavan. 1999. Computing on data streams. In *External Memory Algorithms: DIMACS Workshop External Memory and Visualization, May 20-22, 1998*, Vol. 50. American Mathematical Soc., 107.

Thomas Schank and Dorothea Wagner. 2005. Finding, counting and listing all triangles in large graphs, an experimental study. In *Experimental and Efficient Algorithms*. Springer, 606–609.

Charalampos E Tsourakakis, U Kang, Gary L Miller, and Christos Faloutsos. 2009. Doulion: counting triangles in massive graphs with a coin. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 837–846.