Triangle Listing in Massive Networks

Brita Munsinger
Department of Computer Science
University of Texas at San Antonio

October 27, 2015

Abstract

Triangle listing has many important applications in computer science, such as the analysis of complex networks that emerge from real-world data sets. As data sets have grown larger in recent years, a new challenge for traditional triangle listing algorithms is how to efficiently process a graph that is too large to fit in main memory. This paper presents an algorithm that addresses this issue.

1 Introduction

Triangle listing is the discovery and enumeration of cycles of length three in graphs. There are numerous uses of triangles in the analysis of the structure of graphs and patterns within them, for example in finding dense subgraphs or areas of high connectivity. Recently, large data sets, such as those found within social networks, have become more prevalent.

How to analyze these potentially massive graphs is a real challenge. Traditionally, triangle listing algorithms have been designed to process graphs that can fit in their entirety in main memory. This simplifies the analysis of the graph but limits the scope of the algorithm. Technically an in memory algorithm could be used to analyze a larger graph, but the repeated access to auxiliary storage makes this cost prohibitive in terms of running time.

We have developed an approach that can iteratively partition the graph and keep track of the triangles found. As triangles are found, they are removed from the graph. This continues until all triangles have been found and the input graph is empty. We ensure we do not miss triangles that may cross partition boundaries by augmenting the graph with supplementary data.

2 Related Work

Triangle listing is a rich area of research and there are numerous variations within the field. While our approach focuses on listing all the triangles in a graph, there are applications in which counting the number of triangles is sufficient, for example in calculating the clustering coefficient. Tsourakakis presents an approach for counting triangles using eigenvalues calculated from a matrix representation of the graph. His method uses an in memory algorithm, so it is addressing different challenges than those we explore in this paper.

Another triangle counting algorithm by Suri et al has some common elements to the problem we are exploring. Although their focus is on triangle counting rather than listing, like us, they are concerned with how to handle the analysis of graphs that do not fit in memory. Their approach differs from ours in that they apply MapReduce to the triangle problem in order to parallelize it rather than dividing up the problem and analyzing partitions sequentially as we do. As with Tsourakakis, their work is designed to count triangles, in order to calculate such measures as the clustering coefficient.

Alon et al take a different approach to the triangle counting problem. They explore how to generalize algorithms to count not just triangles, but cycles of

an arbitrary fixed length. As this paper is highly theoretical, issues of whether the graph fits in memory are not relevant to their analysis as they are in our work.

Much research in this area has a theoretical focus. Schank et al provides an alternative perspective on triangle algorithms by implementing several and comparing their performance on generated graphs. In contrast to the other prior work explored here, Schank et al include both counting and listing algorithms in their analysis. They found that no algorithm was best in all circumstances, but that *forward*, a modified version of the standard *edge-iterator* algorithm, was a good compromise.

3 Our Algorithm

At the core of our algorithm is keeping track of triangles as we list them. To facilitate this we define three types of triangles, Type 1, 2 and 3. Type 1 triangles fall entirely within a single partition. Likewise, Type 2 triangles span two partitions and Type 3 triangles span three partitions. In Figure 1, triangle *abc* represents a Type 1 triangle, triangle *bcg* is a Type 2 triangle, and triangle *dej* is a Type 3 triangle.

In order to keep track of edges that span partitions, we introduce the concept of extended subgraph. Additional directed edges are added to nodes that are connected across partition boundaries. Figure 2 shows an example of extended subgraphs.

Our algorithm itself is listed in Figures 3 and 4. The input to our algorithm is the graph and the final output is a list of all triangles in the graph. The algorithm works iteratively, first partitioning the graph, then processing each partition in turn. The subprocedure shown in Algorithm 3 in Figure 4 finds all Type 1 and Type 2 triangles in the subgraph and lists them out. Once the triangles are listed, those edges are removed from the graph.

Type 3 triangles are handled indirectly. As edges are removed, the overall graph shrinks and there will be fewer partitions at each iteration. Eventually, what were once Type 3 triangles spanning multiple partitions will become Type 1 or 2 and be listed by Algorithm 3. Once these partitions have been processed, the algorithm begins again, repartitioning the remaining graph. The algorithm runs until all triangles have been counted, all edges removed, and the graph is empty.

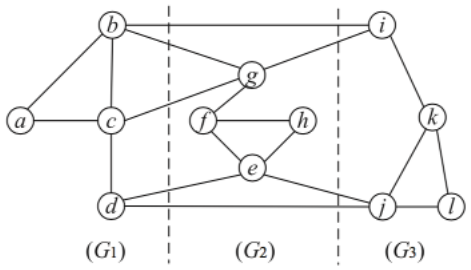

Figure 1: This partitioned graph includes triangles of Types 1, 2 and 3.

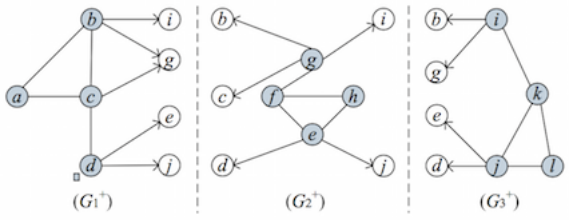

Figure 2: This shows the extended subgraphs for each partition.

4 Conclusions

The algorithm we present addresses the challenge of how to efficiently process a graph that is too large

ALGORITHM 2: I/O-Efficient Triangle Listing

Input: A graph $G = (V_G, E_G)$
Output: $\triangle(G)$

```
1.  while(G is not empty)
2.      Partition G into P = {G_1, ..., G_i, ..., G_p};
3.      for each extended subgraph G_i^+ of G_i ∈ P do
4.          List all triangles in △1(G_i^+) and △2(G_i^+) (by Algorithm 3);
5.          Remove all edges in G_i from G;
        end
    end
```

Figure 3: Our triangle listing algorithm

ALGORITHM 3: Triangle Listing in Extended Subgraph

Input: An extended subgraph $H^+ = (V_{H^+}, E_{H^+})$
Output: $\triangle 1(H^+)$ and $\triangle 2(H^+)$

```
1.   for each u ∈ V_H do
2.       for each v ∈ adj_H(u), where v > u, do
3.           for each w ∈ (adj_H+(u) ∩ adj_H+(v)) do
4.               if(w > v or w ∉ V_H)
5.                   List Δ_uvw;
                 end
             end
         end
     end
```

Figure 4: A subprocedure for algorithm 2

to fit into main memory. Our simulation data shows that our algorithm performs well on graphs too large to fit in memory, and even on graphs that do fit in memory it performs comparably to standard in memory triangle listing algorithms.

5 References

N. Alon, R. Yuster, and U. Zwick, "Finding and Counting Given Length Cycles.,' Algorithmica, vol. 17, 1997, p. 209-223.

S. Chu and J. Cheng, "Triangle Listing in Massive Networks," ACM Transactions on Knowledge Discovery from Data, vol. 6, no. 4, 2012.

T. Schank and D. Wagner, "Finding, Counting and Listing All Triangles in Large Graphs, an Experimental Study," WEA, 2005, p. 606-609.

S. Suri and S. Vassilvitskii, "Counting Triangles and the Curse of the Last Reducer," WWW, 2001.

C. Tsourakakis, "Fast Counting of Triangles in Large Real Networks Without Counting: Algorithms and Laws," Eighth IEEE International Conference on Data Mining, 2008, p. 608-617.