Finding, Counting and Listing all Triangles in Large Graphs, An Experimental Study*

Thomas Schank and Dorothea Wagner

University of Kalrsruhe, Germany

1 Introduction

In the past, the fundamental graph problem of triangle counting and listing has been studied intensively from a theoretical point of view. Recently, triangle counting has also become a widely used tool in network analysis. Due to the very large size of networks like the Internet, WWW, or social networks, the efficiency of algorithms for triangle counting and listing is an important issue. The main intention of this work is to evaluate the practicability of triangle counting and listing in very large graphs with various degree distributions. We give a surprisingly simple enhancement of a well known algorithm that performs best, and makes triangle listing and counting in huge networks feasible. This paper is an extended abstract of [SW05].

2 Definitions

Let $G = (V, E)$ be an undirected, simple graph with a set of nodes V and a set of edges E. We denote the *number of nodes* with n and the *number of edges* with m. The *degree* $d(v) := |\{u \in V : \exists\{v, u\} \in E\}|$ of node v is defined to be the number of nodes in V that are adjacent to v. The *maximal degree* of a graph G is defined as $d_{\max}(G) = \max\{d(v) : v \in V\}$. An *k-clique* is a complete graph with k nodes. Unless otherwise declared we assume graphs to be connected. A *triangle* $\Delta = (V_\Delta, E_\Delta)$ of a graph $G = (V, E)$ is a three node subgraph with $V_\Delta = \{u, v, w\} \subseteq V$ and $E_\Delta = \{\{u, v\}, \{v, w\}, \{w, u\}\} \subseteq E$. We use the symbol $\delta(G)$ to denote the *number of triangles* in graph G. Note that an n-clique has exactly $\binom{n}{3}$ triangles and asymptotically $\delta_{\text{clique}} \in \Theta(n^3)$. In dependency to m we have accordingly $\delta_{\text{clique}} \in \Theta(m^{3/2})$ and by concentrating as many edges as possible into a clique we observe that there exists a family of graphs G_m, such that $\delta(G_m) \in \Theta(m^{3/2})$.

* This work was partially supported by the DFG under grant WA 654/13-2, by the European Commission - Fet Open project COSIN - COevolution and Self-organization In dynamical Networks - IST-2001-33555, and by the EU within the 6th Framework Programme under contract 001907 (integrated project DELIS).

3 Algorithms

We call an algorithm a *counting algorithm*, if it outputs the number of triangles $\delta(v)$ for each node v and, call it a *listing algorithm*, if it outputs the three participating nodes of each triangle. A listing algorithm requires at least one operation per triangle. For the running time we get worst case lower bounds of $\Omega(n^3)$ in terms of n and $\Omega(m^{3/2})$ in terms of m by the final observation in Section 2.

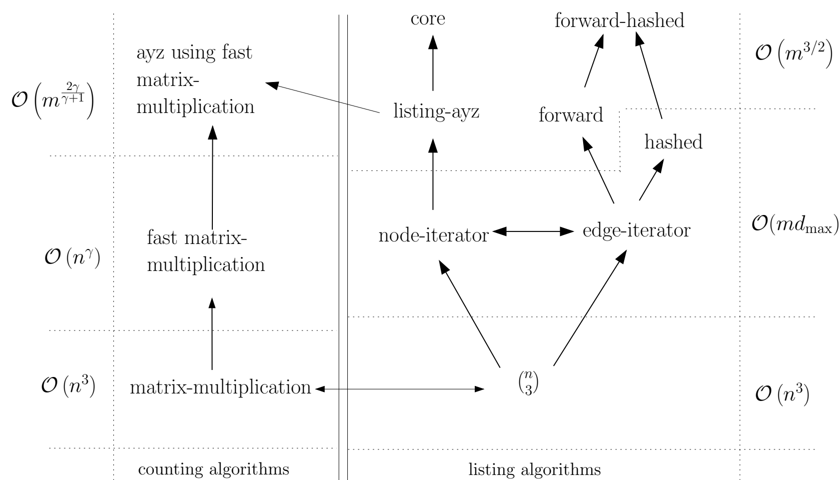


Fig. 1. An overview of the presented algorithms.

Simple approaches are to use matrix multiplication as a counting algorithm or to check for connecting edges between any three nodes as a listing algorithm. Traversing over all nodes and checking for existing edges between any pair of neighbors is part of the folklore. This algorithm, which we call *node-iterator*, has running time $\mathcal{O}\big(nd_{\max}^2\big) \subseteq \mathcal{O}\big(n^3\big)$. The Algorithm *listing-ayz* is the listing version of the currently most efficient counting algorithm [AYZ97]. It has running time in $\mathcal{O}\big(m^{3/2}\big)$. Algorithm *node-iterator-core* uses the concept of cores. It takes a node with currently minimal degree, computes its triangles in the same fashion as in *node-iterator* and then removes the node from the graph. The running time is in $\mathcal{O}\big(nc_{\max}^2\big)$, where $c(v)$ is the core number of node v. Since *node-iterator-core* is an improvement over *listing-ayz* the running time of *node-iterator-core* is also in $\mathcal{O}\big(m^{3/2}\big)$.

Similar to *node-iterator* one can also traverse over all edges and compare the adjacency lists of the two incident nodes. This algorithm, which we call *edge-iterator*, is equivalent to an algorithm introduced by Batagelj and Mrvar [BM01]. The running time without preprocessing is in $\mathcal{O}(md_{\max})$. It can actually be shown that *node-iterator* and *edge-iterator* are asymptotically equivalent, see

[SW05] for details. Algorithm *forward* is an improvement of *edge-iterator*. The pseudo code is listed in Algorithm 1. It can be shown, that *forward* has running time in $\mathcal{O}(m^{3/2})$. Both algorithms can be further improved by certain methods relying on hashing, see [SW05] for details.

Algorithm 1: *forward*

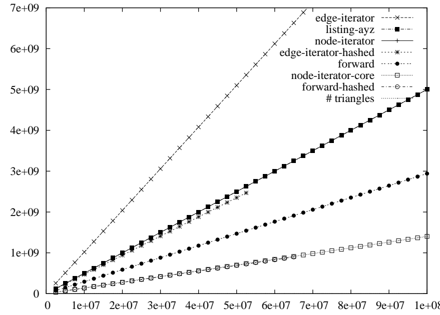
```
Input: ordered list (high degree first) of vertices (1, ..., n); Adjacencies Adj(v)
Data: Node Data: A(v);
for v ∈ V do
    A(v) ← ∅
for s ∈ (1, ..., n) do
    for t ∈ Adj(s) do
        if s < t then
            foreach v ∈ A(s) ∩ A(t) do
                output triangle {v, s, t} ;
            A(t) ⟵ A(t) ∪ {s};
```

4 Experiments

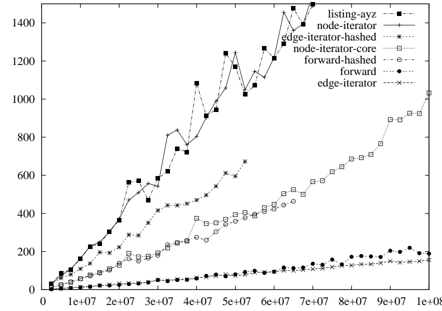
The algorithms are tested in two ways. On the one hand, we list the execution time of the algorithms. Additionally, we give the *number of triangle operations*, which in essence captures the asymptotic running time of the algorithm without preprocessing. The algorithms are implemented in C++. The experiments were carried out on a 64-bit machine with a AMD Opteron Processors clocked at 2.20G-Hz. Figure 2 shows the results on generated $G_{n,m}$ graphs, where m edges are inserted randomly between n nodes. These $G_{n,m}$ graphs tend to have no high degree nodes and to have a very low deviation from the average degree. However, this seems to be not true for many real networks [FFF99]. Therefore, Figure 3 shows results on modified $G_{n,m}$ graphs with $\mathcal{O}(\sqrt{n})$ high degree nodes.

5 Conclusion

The two known standard Algorithms *node-iterator* and *edge-iterator* are asymptoticly equivalent. However, the Algorithm *edge-iterator* can be implemented with a much lower constant overhead. It works very well for graphs where the degrees do not differ much from the average degree. If the degree distribution is skewed refined algorithms are required. The Algorithm *forward* shows to be the best compromise. It is asymptotically efficient and can be implemented to have a low constant factor with respect to execution time.

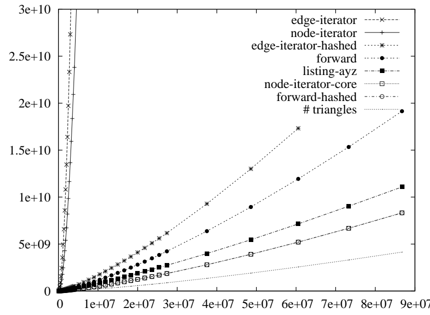


(a) Triangle Operations vs m

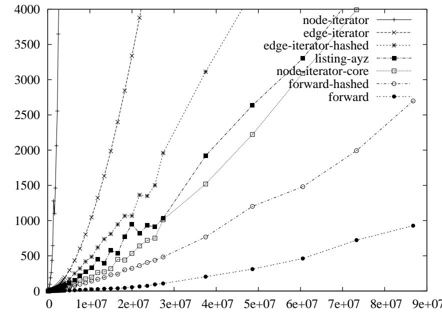


(b) Execution Times (sec.) vs m

Fig. 2. Generated $G_{n,m}$ Graphs



(a) Triangle Operations vs m



(b) Execution Times (sec.) vs m

Fig. 3. Generated Graphs with High Degree Nodes

References

[AYZ97] Noga Alon, Raphael Yuster, and Uri Zwick. Finding and counting given length cycles. *Algorithmica*, 17(3):209–223, 1997.

[BM01] Vladimir Batagelj and Andrej Mrvar. A subquadratic triad census algorithm for large sparse networks with small maximum degree. *Social Networks*, 23:237–243, 2001.

[FFF99] Michalis Faloutsos, Petros Faloutsos, and Christos Faloutsos. On power-law relationships of the Internet topology. In *Proceedings of SIGCOMM'99*, 1999.

[SW05] Thomas Schank and Dorothea Wagner. Finding, counting and listing all triangles in large graphs. Technical report, Universität Karlsruhe, Fakultät für Informatik, 2005.