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

Thomas Schank and Dorothea Wagner

University of Kalrsruhe, Germany

Abstract. A rigorous experimental study of algorithms for counting and listing all triangles in large existing networks as well as generated graphs is presented. In the past, this fundamental graph problem 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.

1 Introduction

Counting the number of triangles in a graph is a fundamental problem with various applications. The problem has been studied before from a theoretic point of view. It can be seen as a special case of counting given length cycles [AYZ97]. Actually, it is a basic ingredient for counting complete subgraphs of given size [KKM00] or other certain small subgraphs, so-called motifs [MSOI+02]. On the other hand, counting triangles is a basic issue in network analysis. Due to the increasing interest in analyzing large networks like the Internet, the WWW or social networks, the computation of network indices based on counting triangles has become an often used tool in network analysis. For example, the so-called clustering coefficient [WS98] is frequently quoted as an important index for measuring the concentration of clusters in graphs respectively its tendency to decompose into communities [EKM+04]. The local clustering coefficient of a node v is defined as the likeliness that two neighbors u and w of v are also connected, while the clustering coefficient of a graph is just the normalized sum of the clustering coefficient of its nodes. Accordingly, its computation involves counting the number of triangles. Similarly the transitivity coefficient

* This is an extend version of [SW05]

** This work was partially supported by the DFG under grant WA 654/13-1, 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).

of a graph [HP57,HK79], which is just three times the number of triangles divided by the number of triples (paths of length two) in the graph is sometimes considered. It was used in [NWS02] for analyzing e.g. the movie actor network considered also in our experimental study. Actually, in [NWS02] it was claimed to be equal to the clustering coefficient which has been disproved [BR02,SW04]. The theoretically most efficient algorithms for counting triangles are based on fast matrix multiplication [CW90]. However fast matrix multiplication is more of theoretic interest. Moreover, algorithms based on matrix multiplication can be only used for triangle counting but not for listing. Alternative approaches can be seen as straight forward iteration over the nodes or edges of the graph like the procedure presented in [BM01], and can also be easily modified to list all triangles.

This paper now presents an intensive experimental evaluation of various algorithms for counting and listing all triangles in a graph. To the best of our knowledge, these algorithms have not been studied comparatively from an experimental point before. Real world graphs as well as generated graphs are considered. The main intention of this work is to evaluate the practicability of triangle counting and listing in very large graphs with various degree distributions. Besides the experimental study, we give an enhancement of a well known algorithm which turns out to perform very well regarding execution time.

2 Definitions and Properties

Let $G = (V, E)$ be an undirected, simple graph with a set of nodes V and a set of edges E. We use the symbol n for the *number of nodes* and the symbol m for the *number of edges*. 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 *$n$-clique* is a complete graph with n 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 $V_\Delta = \{u, v, w\} \subset V$ and $E_\Delta = \{\{u, v\}, \{v, w\}, \{w, u\}\} \subset E$. 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 get the following lemma:

Lemma 1. *There exists a graph G_m with m edges, such that*

$$\delta(G_m) \in \Theta(m^{3/2}).$$

3 Algorithms

We call an algorithm a *counting algorithm* if it outputs the number of triangles $\delta(v)$ for each node v and 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 Lemma 1. A very simple algorithm with running time $\binom{n}{3}$ is to check for edges between nodes of every three element subset of V. There exists a counting algorithm that beats the $\Omega(n^3)$ bound for listing algorithms. If $A(G)$ is the adjacency matrix of graph G, then the diagonal elements of $A(G)^3$ contain two times the number of triangles of the corresponding node. This immediately leads to a counting algorithm with running time $\Theta(n^3)$ respectively $\Theta(n^\gamma)$, where γ is the *matrix multiplication exponent*. It is currently known that $\gamma \leq 2.376$ [CW90].

An algorithm with running time $\Omega(m^{3/2})$ has been proposed by Itai and Rodeh [IR78]. A counting algorithm that beats the $\Omega(m^{3/2})$ bound was given by Alon, Yuster and Zwick [AYZ97]. We will discuss its corresponding listing version among the following algorithms which we evaluate experimentally.

Algorithm "*node-iterator*" Algorithm *node-iterator* iterates over all nodes and tests for each pair of neighbors if they are connected by an edge. The asymptotic running time is given by the expression

$$\sum_{v \in V} \binom{d(v)}{2} \tag{1}$$

and bounded by $\emptyset(nd_{\max}^2)$. This is already a considerable improvement over $\emptyset(n^3)$.

Algorithm "*ayz*" and "*listing-ayz*" The counting algorithm due to Alon, Yuster and Zwick [AYZ97] combines the techniques of the algorithms *node-iterator* and *matrix-multiplication*. Informally the algorithm splits the node set into low degree vertices $V_{\text{low}} = \{v \in V : \mathrm{d}(v) \leq \beta\}$ and high degree vertices $V_{\text{high}} = V \setminus V_{\text{low}}$ where $\beta = m^{\gamma-1/\gamma+1}$. The standard method *node-iterator* is performed on the low degree nodes and (fast) matrix multiplication on the induced subgraph of V_{high}. The running time is in $\emptyset(m^{2\gamma/(\gamma+1)})$ [AYZ97]. We can derive a listing algorithm *listing-ayz* with running time in $\emptyset(m^{3/2})$ by using *node-iterator* also for the induced subgraph of V_{high}, in this case $\beta = \sqrt{m}$.

Algorithm *node-iterator-core* The following modification of *node-iterator* is based on the concept of cores. The *k-core* of a graph is the largest node induced subgraph with minimum degree at least k. The *core number $c(v)$ of a node v* is the maximum k of all cores it belongs to. The *core number of a graph c* is the maximal core number of all of its nodes.

The algorithm *node-iterator-core* 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 bounded by

$$\sum_{v \in V} \binom{c(v)}{2}. \tag{2}$$

Analogous to algorithm *node-iterator* we can bound the running time by $\O(nc^2)$. Note that after removing all nodes v with $c(v) \leq \sqrt{m}$ the remaining graph is a subgraph of the node induced subgraph V_{high} of *listing-ayz*. Hence, this algorithm is an improvement to *listing-ayz* and the running time is in $\O(m^{3/2})$, too.

Algorithm "*edge-iterator*" The algorithm *edge-iterator* iterates over all edges and compares the adjacency data structure of both incident nodes. For an edge $\{u, w\}$ the nodes $\{u, v, w\}$ induce a triangle if and only if node v is present in both adjacency data structures $Adj(u)$ and $Adj(w)$.

If the adjacency data $Adj(v)$ is given in a sorted array, we can compare the two adjacencies for an edge $\{u, w\}$ in $d(u) + d(w)$ time. The sorting can be achieved with a preprocessing step in $\sum_{v \in V} d(v) \ln d(v)$ time. We will have to see in the experimental section how this preprocessing influences the overall execution time. For now we will disregard the preprocessing in the running time which can then be expressed with

$$\sum_{\{u,w\} \in E} d(u) + d(w). \tag{3}$$

As with the previous algorithms we can give some less accurate bounds for the running time which are: $\O(md_{\max}) \subset O(mn)$. Comparing $\O(md_{\max})$ with $\O(nd_{\max}^2)$ of *node-iterator* suggests that *edge-iterator* is an improvement to *node-iterator*. However, this is not true! Consider the following amortized analysis: We split the costs $d(u) + d(w)$ for an edge $\{u, w\}$ in $d(u)$ and $d(w)$ units and assign $d(u)$ to node u and $d(w)$ to node w. In the outer loop each node v is passed $d(v)$ times. Hence, the running time captured by Eq. 3 can be equivalently expressed with

$$\sum_{v \in V} d(v)^2. \tag{4}$$

Corollary 1. *Disregarding preprocessing algorithm* edge-iterator *has the same asymptotic time complexity as algorithm* node-iterator*.*

The algorithm *edge-iterator* is equivalent to an algorithm introduced by Batagelj and Mrvar [BM01]. It has been implemented within the software package Pajek [BM98].

Algorithm "*edge-iterator-hashed*" This algorithm is based on *edge-iterator*. If we use a hashed container for the adjacencies we can ask for every node of the smaller container whether it is present in the larger container in $\O(1)$ time. This leads to a running time asymptotic to

$$\sum_{\{u,w\} \in E} \min\{d(u), d(w)\}. \tag{5}$$

Algorithm "*forward*" This is a refinement of algorithm *edge-iterator*. Instead of an adjacency data structure $Adj(v)$ containing all neighbors of v, a dynamic data structure $A(v)$ is used for which always $|A(v)| \leq |Adj(v)|$. See Alg. 1 for the pseudo code and Figure 1 for an example.

Algorithm 1: *forward*

```
Input: ordered list 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};
```

---

edge	$A(b)$	$A(c)$	$A(d)$	$A(e)$	triangles
1 (a, b)	a				
2 (a, c)	a	a			
3 (a, d)	a	a	a		
4 (a, e)	a	a	a	a	
5 (b, c)	a, b	a	a	a	$\{a, b, c\}$
6 (b, d)	a, b	a	a, b	a	
7 (d, e)	a, b	a	a, b	a, d	$\{a, d, e\}$

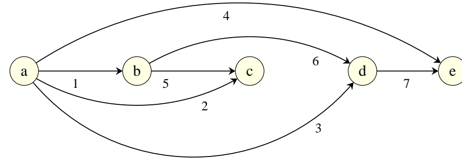


Fig. 1. Example for algorithm *forward*.

As Figure 1 suggest, the algorithm is conveniently regarded on the directed graph induced by the ordering of the nodes. The size of the data structure $A(v)$ is then bounded by the in-degree of node v, and the running time is bounded by the expression

$$\sum_{\{u,w\} \in E} d_{\text{in}}(u) + d_{\text{in}}(w). \tag{6}$$

As a heuristic to minimize Eq. 6 the nodes are considered in decreasing order of their degrees, which implies a preprocessing in $\emptyset(n \log n)$. This ordering suffices to achieve the $\emptyset(m^{3/2})$ bound.

Lemma 2. *The running time of algorithm* forward *is in* $\emptyset(m^{3/2})$.

Proof. Similar to algorithm *edge-iterator* we can bound the running time by $\emptyset(\max\{d_{\text{in}}\}m)$. We now show that $d_{\text{in}}(v) \in \emptyset(\sqrt{m})$ for all nodes $v \in V$. Assume that there exists a node v with $d(v) > \sqrt{m}$. Consider the vertices $v_1, \ldots, v_n$ in decreasing order of their degrees and let k be the index where $d(v_k) > \sqrt{m}$ but $d(v_{k+1}) \leq \sqrt{m}$. From this it follows that $k\sqrt{m} \leq \sum_{i \leq k} d(v_i)$. On the other hand the handshaking lemma gives $\sum_{i \leq k} d(v_i) \leq \sum_{i \leq n} d(v_i) = 2m$. It follows that $k \leq 2\sqrt{m}$ and we can conclude that $d_{\text{in}}(v_i) \leq 2\sqrt{m}$ for $i \leq k$, too.

Algorithm "*forward-hashed*" We can combine the methods of *forward* with *edge-iterator-hashed* which gives us an improved asymptotic running time

$$\sum_{\{u,w\} \in E} \min\{d_{\text{in}}(u), d_{\text{in}}(w)\}. \tag{7}$$

3.1 Overview of the Algorithms

Figure 2 shows an overview of the presented algorithms with their guaranteed running time. The interesting listing algorithms will be experimentally compared in the next section.

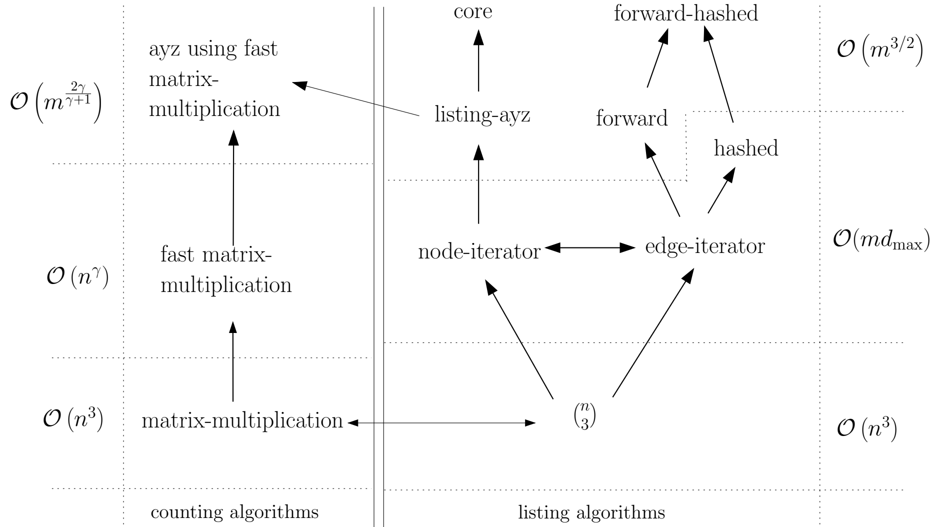


Fig. 2. An overview of the presented algorithms. The running times do not include preprocessing.

4 Implementation and Experiments

The algorithms are implemented in C++ using the *STL* and compiled with the *gnu g++* compiler version 3.4 with Options "-g -O2". The experiments were carried out on a 64-bit machine with two AMD Opteron Processors clocked at 2.20GHz. The implemented algorithms only used one of the processors and were terminated if they used more than 6GB of memory. The execution times of the implemented algorithms were measured in seconds using the *getrusage* function.

The graphs are represented using node pointers of std::vector type for the nodes and their adjacency data structure. The algorithms *node-iterator*, *listing-ayz* and *node-iterator-core* use hashing for testing edge existence in constant time. The hash function combines two random numbers of size size_t with XOR, one for each node. The __gnu_cxx::hash_map was used as a hash container. Creating the random bits is not contained in the execution time of the experiments, whereas filling the container is included. We also compared the performance between a hashed container and a balanced binary tree. There seems to be no substantial asymptotic difference, but the hashed container performed generally better (see Appendix C).

A straight forward implementation of *node-iterator-core* as mentioned in Section 3 requires multiple linked data structures which turned out to consume too much memory. We used an equivalent static algorithm instead. The algorithms *edge-iterator* and *forward* require to find common nodes in two adjacency data structures. For that purpose the used std::vector containers are sorted in a preprocessing step using the std::sort function. The sorting is included in the execution time of the algorithms. The generation of the ordering for *forward* is also included in the execution time.

4.1 Experiments

The implemented algorithms are evaluated on several networks originating from real world as well as on generated graphs. We will show some selected results here (a more complete listing can be found in Appendix A). 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 kind of operation considered as a triangle operation differs for the various algorithms but in all cases it represents the number of triangle tests, e.g. for the algorithm *node-iterator* the number of triangle operations is equal to $\sum_{v \in V} \binom{d(v)}{2}$ and for the *edge-iterator* equal to $\sum_{v \in V} d(v)^2$.

Road Network Germany This network is based on the roads in Germany. Hence, it is very sparse, almost planar, has very low average and maximum degree and in consequence a very low deviation from the average degree, see Table 1(b). The std. deviation from the average degree is an interesting measure. The square of it gives an indication how much improvement is possible for the

more refined algorithms compared to the two standard algorithms *node-iterator* and *edge-iterator*. The performance of the algorithms is listed in Table 1(a). As expected algorithm *edge-iterator* has the highest asymptotic effort in the number of triangle operations. However, in terms of execution time it outperforms all the other algorithms. The two algorithms *edge-iterator-hashed* and *forward-hashed*, which use hashed data structures for every node, perform badly. (See also Appendix A.6 for a visualization of the results.)

algorithm	triangle operations	execution time in seconds
node-iterator	10752498	13.73
listing-ayz	10752498	13.95
node-iterator-core	1244194	11.99
edge-iterator	33398998	4.27
edge-iterator-hashed	14476855	55.47
forward	1786531	9.03
forward-hashed	1426197	43.97

(a) Algorithm Performance

nodes	4799875
edges	5947001
$d_{\max}$	7
$d_{\min}$	1
d_{avr}	2.5
d stddeviation	0.90
core number	3
triples	10752498
triangles	172699
transitivity	0.048
clustering coefficient	0.050

(b) Graph Properties

Table 1. Road Network of Germany

Movie Actor Network 2004 This graph is constructed from *The Internet Movie Database* of the year 2004. Each node represents an actor. Two actors share a link if and only if they ever played together in a movie. This network has a more interesting structure compared to the network of Section 4.1. The degree distribution is somewhat skewed with a std. deviation of 183 from the average degree, see Table 2(b). The performance of the algorithms is shown in Table 2(a). The algorithms *node-iterator-core* and *forward-hashed* are very efficient with respect to the number of triangle operations. As in Section 4.1 the algorithm *listing-ayz* is no improvement to algorithm *node-iterator*, since $\sqrt{m} = 5252$ is higher than the maximum degree. Again *edge-iterator* performs relatively well in execution time considering highest number of triangle operations. The implementation of algorithm *forward* performs best in execution time. (See also Appendix A.2 for a more visual representation of Table 2. See Appendix A.1 and A.3 for related networks.)

4.2 Generated $G_{n,m}$ Graphs

To show how the performance of the algorithms evolves for graphs of increasing size, we generated a series of random graphs. Figure 3 shows the results on

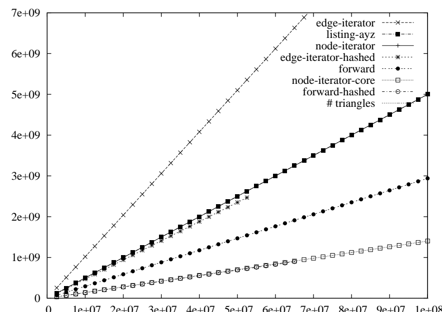
algorithm	triangle operations	execution t. in seconds
node-iterator	13452269555	4649.05
listing-ayz	13452269555	5285.50
node-iterator-core	1725685526	610.61
edge-iterator	26959701660	174.04
edge-iterator-hashed	7460874664	1377.73
forward	5423623560	64.48
forward-hashed	1745104092	504.26

(a) Algorithm Performance

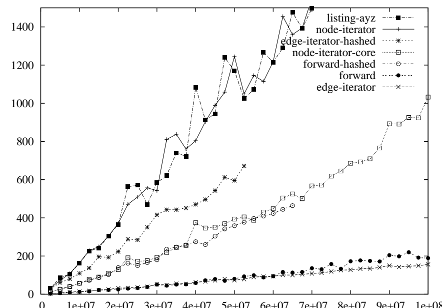
nodes	667609
edges	27581275
d_{max}	4605
d_{min}	1
d_{avr}	82.6
d $_{\text{stddeviation}}$	183.18
core number	1005
triples	13452269555
triangles	1176613576
transitivity	0.262
clustering coefficient	0.796

(b) Graph Properties

Table 2. Movie Actor Network 2004



(a) Triangle Operations vs Numb. of Edges



(b) Execution Times (sec.) vs Numb. of Edges

Fig. 3. Generated $G_{n,m}$ Graphs

generated graphs where m edges are inserted randomly between n nodes. We do not allow loops or multiple links in our $G_{n,m}$ graphs. In this case we consider graphs where the relation of the parameters n and m is chosen such that the average degree equals 50.

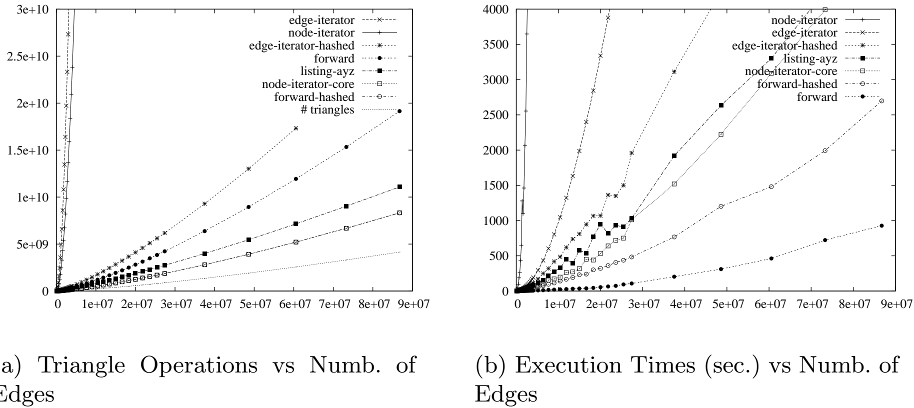
Figure 3(a) shows the number of triangle operations versus the number of edges. Since $G_{n,m}$ graphs are very unlikely to have high degree nodes, algorithm *listing-ayz* and *node-iterator* perform equally well with respect to the number of triangle operations. The algorithms *node-iterator-core* and *forward-hashed* do most efficiently limit the number of triangle operations. They are very close to the optimum, i.e. the number of triangles.

Figure 3(b) shows the execution time in seconds versus the number of edges. Again the algorithm *edge-iterator* performs best together with *forward*. Both algorithms are very simple and do not use complicated data structures. In contrast the use of hashing slows down the execution time of the corresponding algorithms. The plots of algorithms using hashing show some notable pikes. They are caused by automatic rehashing of the container.

4.3 Generated Graphs with High Degree Nodes

The $G_{n,m}$ graphs like in Section 4.2 tend to have no high degree nodes and to have a very low deviation from the average degree in general. However, many real world networks show a different behavior, see Section 4.1 (and also Appendix A.10) for an example. The famous power law distributions in the degree found in many networks [FFF99] actually hint that skewed degree distributions are very common.

We use a very simple modification of the $G_{n,m}$ model to achieve high degree nodes. As in Section 4.2 we first generate a prime graph G_{n,m_0} which we extend to a graph $G_{n,m,h}$. For each node $1 \leq i \leq h$ we add links to randomly selected nodes until the degree of the i-th node is $\frac{n}{2}\frac{h-i}{h}$. For the results shown in Figure 4 we set the parameter $h = \sqrt{n}$.



(a) Triangle Operations vs Numb. of Edges

(b) Execution Times (sec.) vs Numb. of Edges

Fig. 4. Generated Graphs with High Degree Nodes

Now, the number of triangle operations is obviously not linear in the graph size, see Figure 4(a). Again the algorithms *node-iterator-core* and *forward-hashed* perform best in limiting the asymptotic effort. However they are only in second and third place in the execution time. Algorithm *forward* performs clearly best.

5 Conclusion

The two known standard algorithms *node-iterator* and *edge-iterator* are asymptoticly equivalent (Lemma 1). However, the algorithm *edge-iterator* can be implemented with a much lower constant overhead (Figure 3(b)). It works very well for graphs where the degrees do not differ much from the average degree (Figure 3(b), Table 1).

If the degree distribution is skewed refined algorithms are required. The algorithms *node-iterator-core* and *forward-hashed* are most efficient in reducing the number of triangle operations (Table 1(a), 2(a), Figure 3(a), 4(a)). However, they require advanced data structures which experimentally result in a high constant overhead(Table 1(a), 2(a), Figure 3(b), 4(b)). The algorithm *forward* experimentally shows to be a good compromise. Its execution time is close to the one of algorithm *edge-iterator* for networks with low deviation from the average degree (Table 1(a), Figure 3(b)). However, it clearly performs best for graphs with notably skewed degree distributions (Table 2(a), Figure 4(b)). In general algorithm *forward* achieves the best execution times. Altogether, we have shown that listing and counting triangles can be performed in reasonable time even for huge graphs.

Acknowledgments

We thank Ulrik Brandes, Sabine Cornelsen and Thomas Willhalm for helpful discussions. Further we would like to thank Vladimir Batagelj, for making available many of the considered real wold networks to us.

References

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

BM98. Vladimir Batagelj and Andrej Mrvar. Pajek – A program for large network analysis. *Connections*, 21(2):47–57, 1998.

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.

BR02. Béla Bollobás and Oliver M. Riordan. Mathematical results on scale-free random graphs. In Stefan Bornholdt and Heinz Georg Schuster, editors, *Handbook of Graphs and Networks: From the Genome to the Internet*, pages 1–34. Wiley-VCH, 2002.

CW90. Don Coppersmith and Shmuel Winograd. Matrix multiplication via arithmetic progressions. *Journal of Symbolic Computation*, 9(3):251–280, 1990.

EKM+04. Stephen Eubank, V.S. Anil Kumar, Madhav V. Marathe, Aravind Srinivasan, and Nan Wang. Structural and algorithmic aspects of massive social networks. In *Proceedings of the 14th Annual ACM–SIAM Symposium on Discrete Algorithms (SODA'04)*, pages 718–727, 2004.

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

HK79. Frank Harary and Helene J. Kommel. Matrix measures for transitivity and balance. *Journal of Mathematical Sociology*, 6:199–210, 1979.

HP57. Frank Harary and Herbert H. Paper. Toward a general calculus of phonemic distribution. *Language : Journal of the Linguistic Society of America*, 33:143–169, 1957.

IR78. Alon Itai and Michael Rodeh. Finding a minimum circuit in a graph. *SIAM Journal on Computing*, 7(4):413–423, 1978.

KKM00. Ton Kloks, Dieter Kratsch, and Haiko Müller. Finding and counting small induced subgraphs efficiently. *Information Processing Letters*, 74(3–4):115–121, 2000.

MSOI+02. Ron Milo, Shai Shen-Orr, Shalev Itzkovitz, Nadav Kashtan, Dmitri Chklovskii, and Uri Alon. Network motifs: Simple building blocks of complex networks. *Science*, 298:824–827, October 2002.

NWS02. Mark E. J. Newman, Duncan J. Watts, and Steven H. Strogatz. Random graph models of social networks. *Proceedings of the National Academy of Science of the United States of America*, 99:2566–2572, 2002.

SW04. Thomas Schank and Dorothea Wagner. Approximating clustering-coefficient and transitivity. Technical Report 2004-9, Universität Karlsruhe, Fakultät für Informatik, 2004.

SW05. Thomas Schank and Dorothea Wagner. Finding, counting and listing all triangles in large graphs, an experimental study. In *Proceedings on the 4th International Workshop on Experimental and Efficient Algorithms (WEA'05)*, volume 3503 of *Lecture Notes in Computer Science*. Springer-Verlag, 2005.

WS98. Duncan J. Watts and Steven H. Strogatz. Collective dynamics of “small-world” networks. *Nature*, 393:440–442, 1998.

Appendix

A Experiments on "real world" Networks

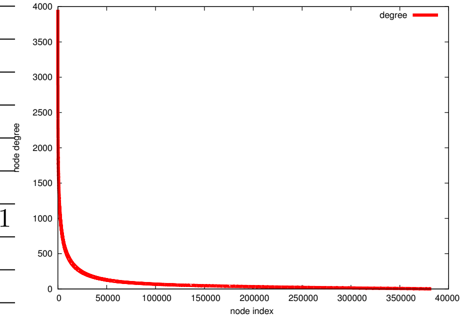
We list experiments on several "real world" networks, meaning that the graph corresponds to some database or existing network. If the original network is directed we consider its underlying undirected network. Potential multiple edges and loops have also been removed. The first row shows some graph properties in the table on the left. The plot on the right shows the nodes with their corresponding degree. The middle row captures the running of the algorithm in seconds. The numeric values are shown in the table on the left. The right hand side shows a visualization. The last row shows the triangle operations of the algorithms as numeric values on the left and visualized on the right.

A.1 Movie Actor Network 2002

This graph is constructed from *The Internet Movie Database* of the year 2002. Each node represents an actor. Two actors share a link if and only if they ever played together in a movie.

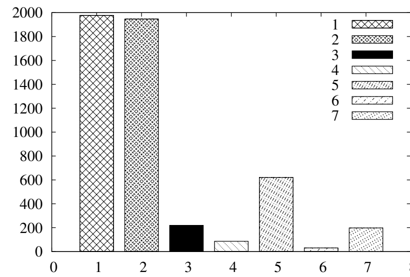
Graph Properties

nodes	382219
edges	15038083
d_{max}	3956
d_{min}	1
d_{avr}	78.7
$d_{\text{stddeviation}}$	163.33
core number	365
triples	6266209411
triangles	346813199
transitivity	0.166
clustering coefficient	0.785



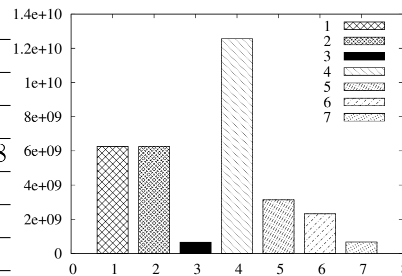
Running Times

	algorithm	seconds
1	*node-iterator*	1976.85
2	*listing-ayz*	1946.62
3	*node-iterator-core*	219.00
4	*edge-iterator*	86.31
5	*edge-iterator-hashed*	619.75
6	*forward*	30.66
7	*forward-hashed*	197.82



Triangle Operations

	algorithm	operations
1	*node-iterator*	6266209411
2	*listing-ayz*	6242906011
3	*node-iterator-core*	663016021
4	*edge-iterator*	12562494988
5	*edge-iterator-hashed*	3134166441
6	*forward*	2320614045
7	*forward-hashed*	671726581

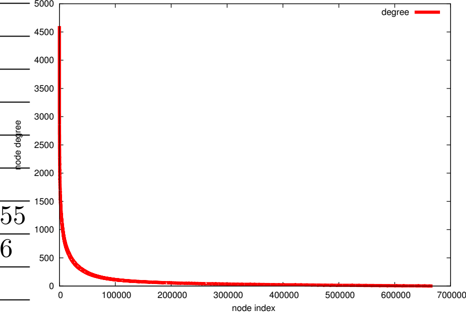


A.2 Movie Actor Network 2004

This is essentially the same as in Sec. A.1 but the database is from the year 2004
.

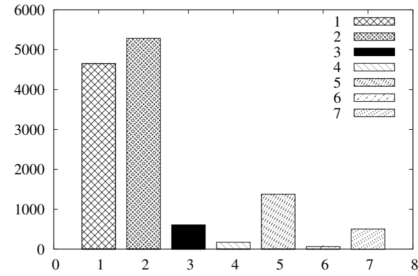
Graph Properties

nodes	667609
edges	27581275
d_{max}	4605
d_{min}	1
d_{avr}	82.6
$d_{\text{ stddeviation}}$	183.18
core number	1005
triples	13452269555
triangles	1176613576
transitivity	0.262
clustering coefficient	0.796



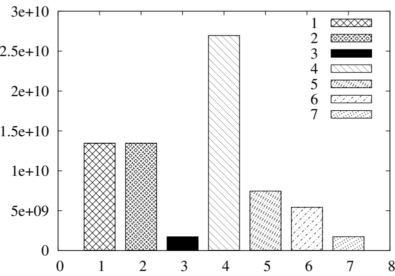
Running Times

	algorithm	seconds
1	*node-iterator*	4649.05
2	*listing-ayz*	5285.50
3	*node-iterator-core*	610.61
4	*edge-iterator*	174.04
5	*edge-iterator-hashed*	1377.73
6	*forward*	64.48
7	*forward-hashed*	504.26



Triangle Operations

	algorithm	operations
1	*node-iterator*	13452269555
2	*listing-ayz*	13452269555
3	*node-iterator-core*	1725685526
4	*edge-iterator*	26959701660
5	*edge-iterator-hashed*	7460874664
6	*forward*	5423623560
7	*forward-hashed*	1745104092

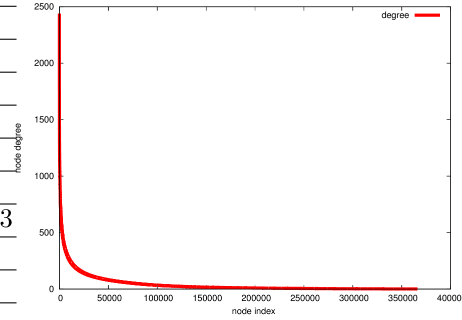


A.3 Movie Actor Network 2004 (no porn)

This is based on the same data like Sec. A.2, but movies were removed if the tile was hinting to pornographic content.

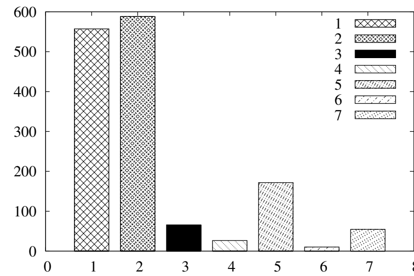
Graph Properties

nodes	366131
edges	7711904
d_{max}	2440
d_{min}	1
d_{avr}	42.1
$d_{\text{ stddeviation}}$	94.30
core number	173
triples	1944966433
triangles	87296640
transitivity	0.135
clustering coefficient	0.765



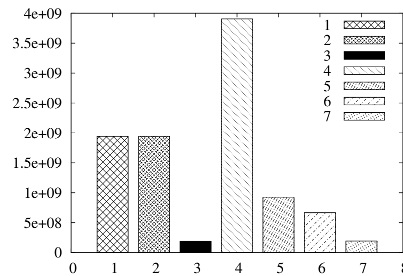
Running Times

	algorithm	seconds
1	*node-iterator*	557.20
2	*listing-ayz*	588.25
3	*node-iterator-core*	65.98
4	*edge-iterator*	26.63
5	*edge-iterator-hashed*	171.88
6	*forward*	10.41
7	*forward-hashed*	54.63



Triangle Operations

	algorithm	operations
1	*node-iterator*	1944966433
2	*listing-ayz*	1944966433
3	*node-iterator-core*	190982368
4	*edge-iterator*	3905356674
5	*edge-iterator-hashed*	926335270
6	*forward*	666275428
7	*forward-hashed*	192700452

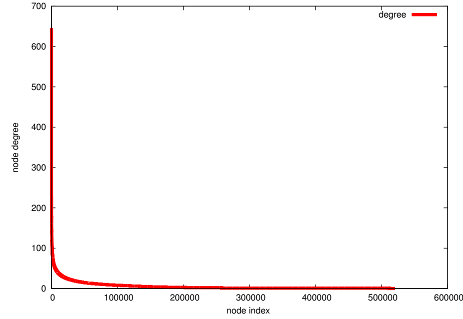


A.4 Notre Dame Actors

This network is also constructed from the *The Internet Movie Database*. However, a node represents an actor or a movie. An edge $\{a, m\}$ exist if and only if an actor a played in a movie m. This network is bipartite and consequently has no triangles. The underlying database is from the year 1999 or earlier.

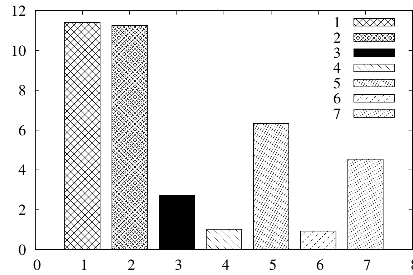
Graph Properties

nodes	520223
edges	1470404
d_{max}	646
d_{min}	0
d_{avr}	5.7
$d_{\text{stddeviation}}$	11.20
core number	14
triples	39484087
triangles	0
transitivity	0.000
clustering coefficient	0.000



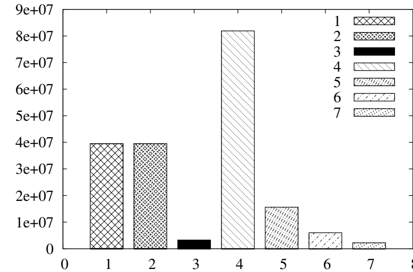
Running Times

	algorithm	seconds
1	*node-iterator*	11.40
2	*listing-ayz*	11.25
3	*node-iterator-core*	2.72
4	*edge-iterator*	1.02
5	*edge-iterator-hashed*	6.33
6	*forward*	0.93
7	*forward-hashed*	4.54



Triangle Operations

	algorithm	operations
1	*node-iterator*	39484087
2	*listing-ayz*	39484087
3	*node-iterator-core*	3271809
4	*edge-iterator*	81908982
5	*edge-iterator-hashed*	15613702
6	*forward*	6011908
7	*forward-hashed*	2227792

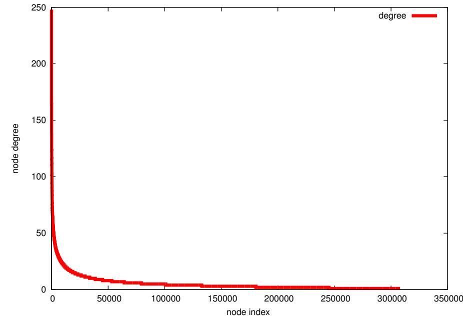


A.5 Authors

The network is based on the *Computer Science Bibliography* at the University of Trier. Two scientists are connected if they are co-author of a publication. The underlying database is from the year 2004.

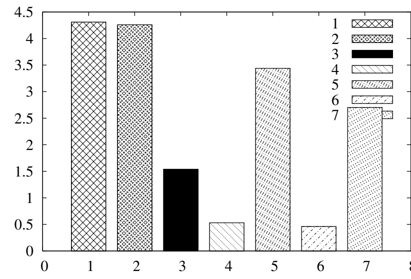
Graph Properties

nodes	307971
edges	831557
$d_{\max}$	248
$d_{\min}$	1
d_{avr}	5.4
$d_{\text{stddeviation}}$	8.44
core number	114
triples	14633230
triangles	1665486
transitivity	0.341
clustering coefficient	0.760



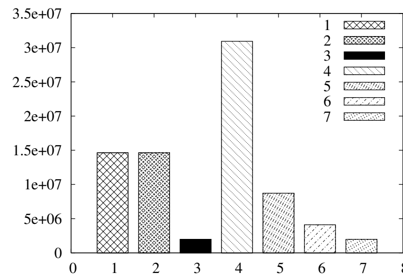
Running Times

	algorithm	seconds
1	*node-iterator*	4.31
2	*listing-ayz*	4.26
3	*node-iterator-core*	1.54
4	*edge-iterator*	0.53
5	*edge-iterator-hashed*	3.44
6	*forward*	0.46
7	*forward-hashed*	2.70



Triangle Operations

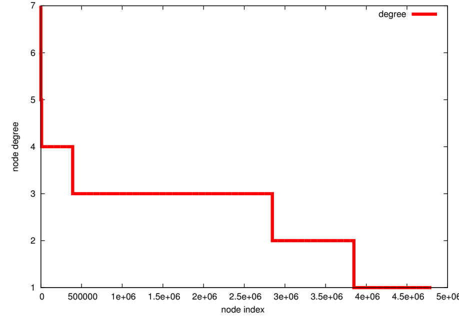
	algorithm	operations
1	*node-iterator*	14633230
2	*listing-ayz*	14633230
3	*node-iterator-core*	1976191
4	*edge-iterator*	30929574
5	*edge-iterator-hashed*	8718461
6	*forward*	4104865
7	*forward-hashed*	1971742



A.6 Road Network Germany

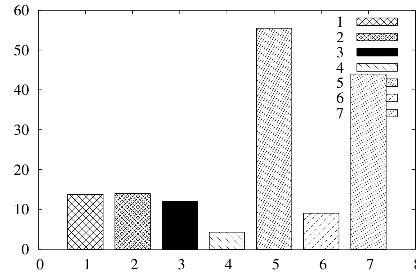
Graph Properties

nodes	4799875
edges	5947001
d_{max}	7
d_{min}	1
d_{avr}	2.5
d stddeviation	0.90
core number	3
triples	10752498
triangles	172699
transitivity	0.048
clustering coefficient	0.050



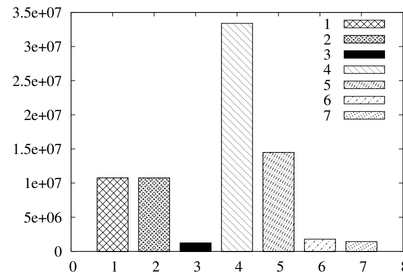
Running Times

	algorithm	seconds
1	*node-iterator*	13.73
2	*listing-ayz*	13.95
3	*node-iterator-core*	11.99
4	*edge-iterator*	4.27
5	*edge-iterator-hashed*	55.47
6	*forward*	9.03
7	*forward-hashed*	43.97



Triangle Operations

	algorithm	operations
1	*node-iterator*	10752498
2	*listing-ayz*	10752498
3	*node-iterator-core*	1244194
4	*edge-iterator*	33398998
5	*edge-iterator-hashed*	14476855
6	*forward*	1786531
7	*forward-hashed*	1426197

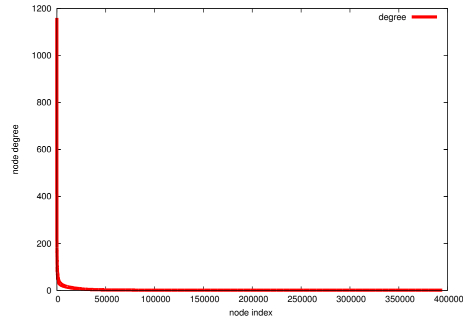


A.7 Google Contest 2002

This network is from the Google contest of the year 2002.

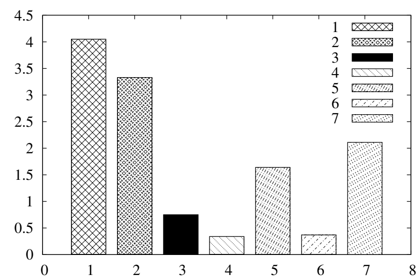
Graph Properties

nodes	394510
edges	480259
$d_{\max}$	1160
$d_{\min}$	1
d_{avr}	2.4
$d_{\text{stddeviation}}$	9.32
core number	13
triples	17834734
triangles	43888
transitivity	0.007
clustering coefficient	0.228



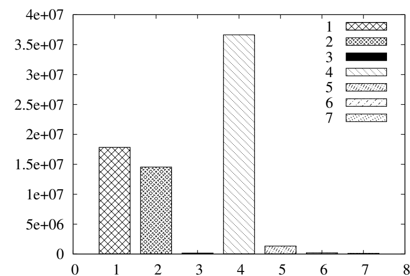
Running Times

	algorithm	seconds
1	*node-iterator*	4.05
2	*listing-ayz*	3.33
3	*node-iterator-core*	0.75
4	*edge-iterator*	0.34
5	*edge-iterator-hashed*	1.64
6	*forward*	0.37
7	*forward-hashed*	2.11



Triangle Operations

	algorithm	operations
1	*node-iterator*	17834734
2	*listing-ayz*	14544261
3	*node-iterator-core*	166977
4	*edge-iterator*	36629986
5	*edge-iterator-hashed*	1342042
6	*forward*	209924
7	*forward-hashed*	118037

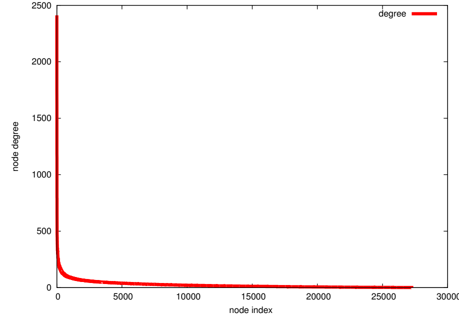


A.8 HEP Literature

Citation data from KDD Cup 2003, a knowledge discovery and data mining competition. The graph is based on a publication data base on High Energy Particle Physics. Two publications are linked if one of them cites the other.

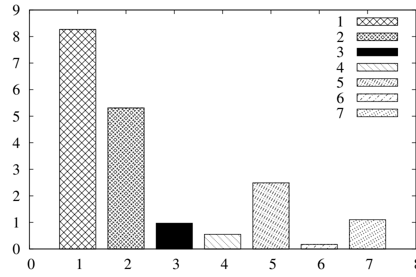
Graph Properties

nodes	27240
edges	341923
d_{max}	2411
d_{min}	0
d_{avr}	25.1
$d_{\text{stddeviation}}$	44.87
core number	37
triples	35665330
triangles	1423613
transitivity	0.120
clustering coefficient	0.330



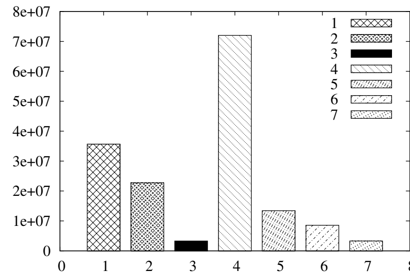
Running Times

	algorithm	seconds
1	*node-iterator*	8.27
2	*listing-ayz*	5.31
3	*node-iterator-core*	0.97
4	*edge-iterator*	0.55
5	*edge-iterator-hashed*	2.49
6	*forward*	0.17
7	*forward-hashed*	1.10



Triangle Operations

	algorithm	operations
1	*node-iterator*	35665330
2	*listing-ayz*	22709903
3	*node-iterator-core*	3297336
4	*edge-iterator*	72014506
5	*edge-iterator-hashed*	13415279
6	*forward*	8552106
7	*forward-hashed*	3295650

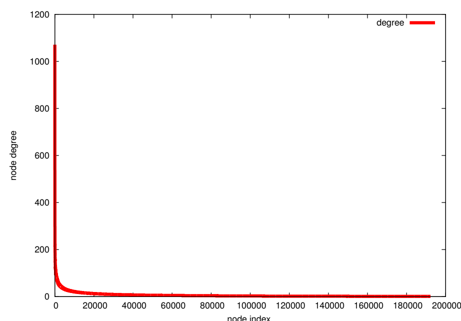


A.9 Router Network

Internet routers (nodes) and their connections (edges) collected by the *Cooperative Association for Internet Data Analysis* (CAIDA). The data is from 2004. Unfortunately the link has either been moved or the data completely removed.

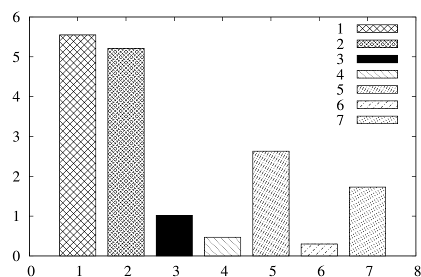
Graph Properties

nodes	192244
edges	609066
$d_{\max}$	1071
$d_{\min}$	1
d_{avr}	6.3
$d_{\text{stddeviation}}$	14.14
core number	32
triples	22468727
triangles	455062
transitivity	0.061
clustering coefficient	0.202



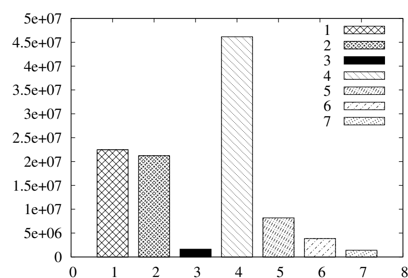
Running Times

	algorithm	seconds
1	*node-iterator*	5.55
2	*listing-ayz*	5.21
3	*node-iterator-core*	1.02
4	*edge-iterator*	0.47
5	*edge-iterator-hashed*	2.63
6	*forward*	0.30
7	*forward-hashed*	1.73



Triangle Operations

	algorithm	operations
1	*node-iterator*	22468727
2	*listing-ayz*	21221322
3	*node-iterator-core*	1636761
4	*edge-iterator*	46155586
5	*edge-iterator-hashed*	8190585
6	*forward*	3853533
7	*forward-hashed*	1411673

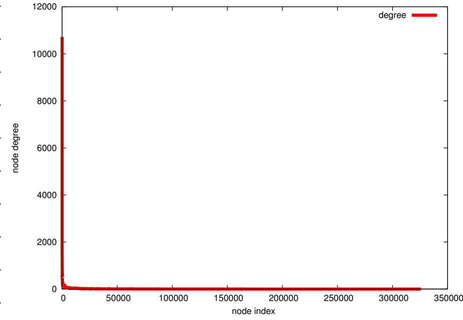


A.10 Notre Dame WWW

Each node represents a web page within the nd.edu domain. An edge between to web pages exist if one of them was linking the other.

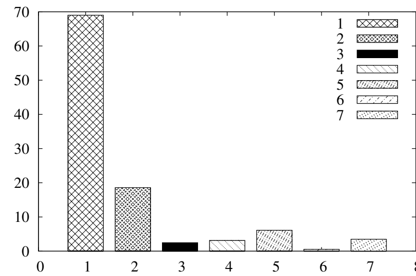
Graph Properties

nodes	325729
edges	1090108
$d_{\max}$	10721
$d_{\min}$	1
d_{avr}	6.7
d stddeviation	42.82
core number	155
triples	304881174
triangles	8910005
transitivity	0.088
clustering coefficient	0.466



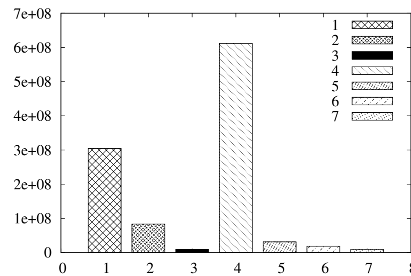
Running Times

	algorithm	seconds
1	*node-iterator*	69.00
2	*listing-ayz*	18.58
3	*node-iterator-core*	2.47
4	*edge-iterator*	3.12
5	*edge-iterator-hashed*	6.11
6	*forward*	0.53
7	*forward-hashed*	3.49



Triangle Operations

	algorithm	operations
1	*node-iterator*	304881174
2	*listing-ayz*	83374735
3	*node-iterator-core*	9664634
4	*edge-iterator*	611942564
5	*edge-iterator-hashed*	31373326
6	*forward*	18742108
7	*forward-hashed*	9473218

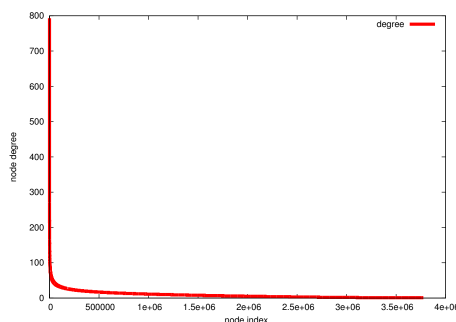


A.11 US Patents

The network Patents is based on the *The NBER U.S. Patent Citations Data File*, version 2001.

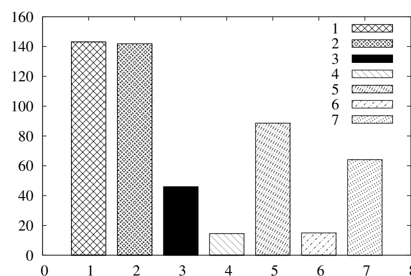
Graph Properties

nodes	3774768
edges	16518947
d_{max}	793
d_{min}	1
d_{avr}	8.8
$d_{\text{stddeviation}}$	10.49
core number	64
triples	335781273
triangles	7515023
transitivity	0.067
clustering coefficient	0.092



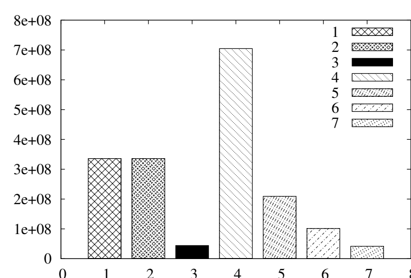
Running Times

	algorithm	seconds
1	*node-iterator*	143.12
2	*listing-ayz*	141.91
3	*node-iterator-core*	46.02
4	*edge-iterator*	14.54
5	*edge-iterator-hashed*	88.64
6	*forward*	14.98
7	*forward-hashed*	64.12



Triangle Operations

	algorithm	operations
1	*node-iterator*	335781273
2	*listing-ayz*	335781273
3	*node-iterator-core*	44077584
4	*edge-iterator*	704600440
5	*edge-iterator-hashed*	209064247
6	*forward*	101235960
7	*forward-hashed*	41627744

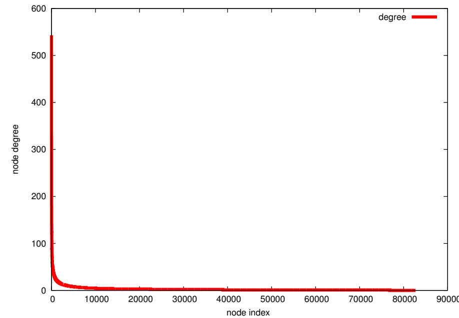


A.12 Wordnet

This network is based on a dictionary.

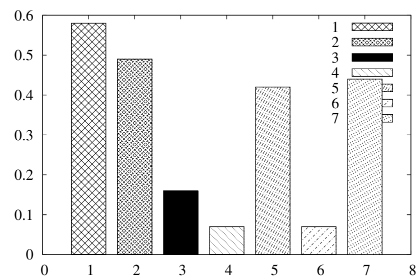
Graph Properties

nodes	82670
edges	120399
$d_{\max}$	543
$d_{\min}$	0
d_{avr}	2.9
d stddeviation	7.74
core number	5
triples	2707088
triangles	5266
transitivity	0.006
clustering coefficient	0.056



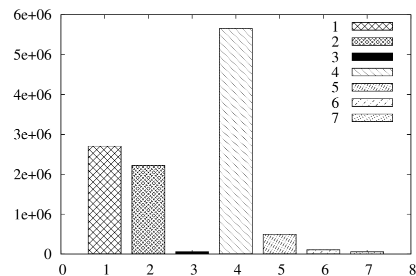
Running Times

	algorithm	seconds
1	*node-iterator*	0.58
2	*listing-ayz*	0.49
3	*node-iterator-core*	0.16
4	*edge-iterator*	0.07
5	*edge-iterator-hashed*	0.42
6	*forward*	0.07
7	*forward-hashed*	0.44



Triangle Operations

	algorithm	operations
1	*node-iterator*	2707088
2	*listing-ayz*	2223834
3	*node-iterator-core*	59299
4	*edge-iterator*	5654974
5	*edge-iterator-hashed*	497430
6	*forward*	106790
7	*forward-hashed*	56875



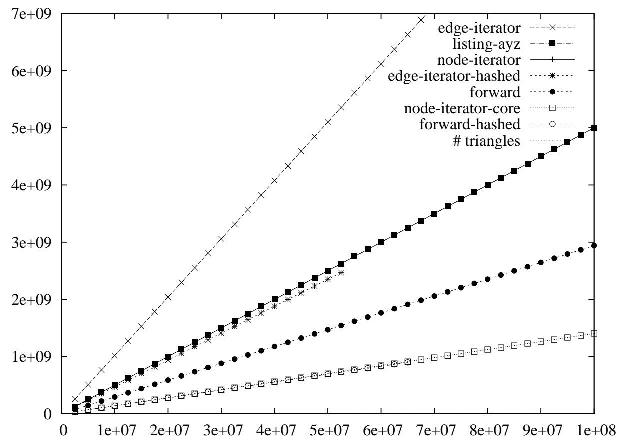
B Experiments on Generated Graphs

We use several series of generated graphs to see how the execution times and the number of triangle operations evolve. The basic graphs $G_{n,m}$ are generated by adding m randomly chosen links between the n nodes. We do not allow loops or multiple links. In this kind of graphs nodes with high degrees are very unlikely. To achieve high degree nodes we extend a prime graph G_{n_0,m_0} to a graph $G_{n,m,h}$ by selecting h nodes for which we add links to randomly selected nodes until the degree of the i-th node is $\frac{n}{2}\frac{h-i}{h}$. Again, we do not allow loops or multiple links.

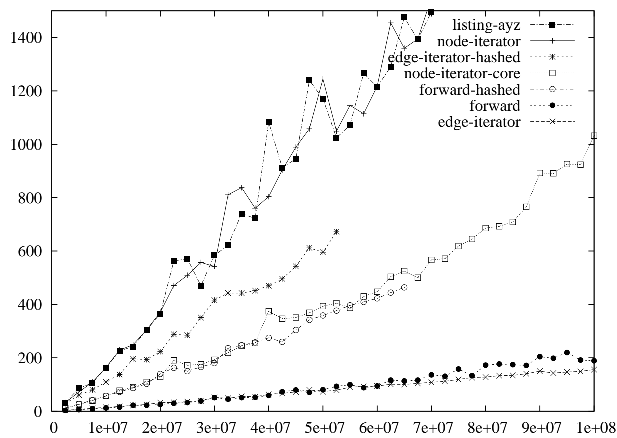
B.1 Sparse $G_{n,m}$

The graphs are generated according to the $G_{n,m}$ model. The relation between n and m was chosen such that the graphs have an average degree of 50.

triangle operations (y-axis) versus number of edges (x-axis)



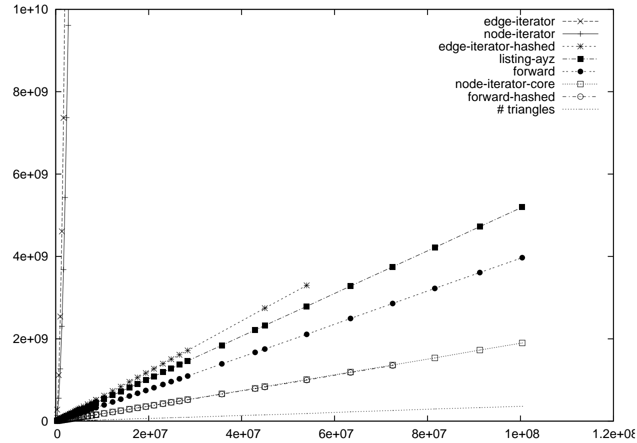
execution time in seconds (y-axis) versus number of edges (x-axis)



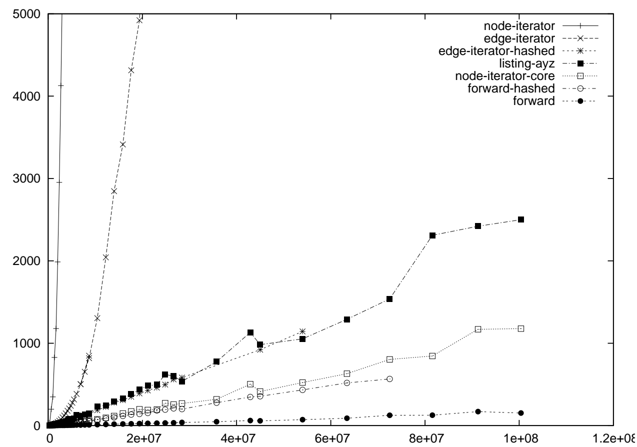
B.2 Sparse $G_{n,m}$ overlayed with $\Theta(\ln n)$ high degree nodes

First a graph G_{n,m_0} with average degree of 50 is generated. This prime graph is extended to a $G_{n,m,h}$ graph with parameter $h = 3 \ln n$.

triangle operations (y-axis) versus number of edges (x-axis)



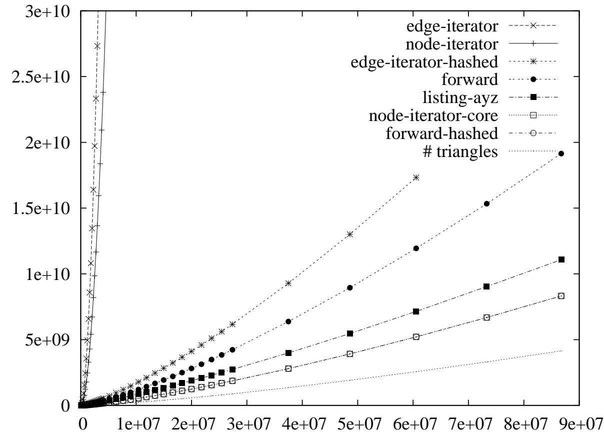
execution time in seconds (y-axis) versus number of edges (x-axis)



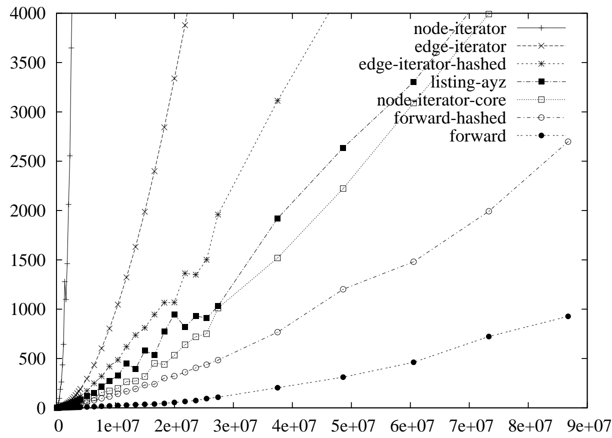
B.3 Sparse $G_{n,m}$ overlayed with $\Theta(\sqrt{n})$ high degree nodes

The graphs are constructed like in Section B.1, but here the parameter h is set to $\sqrt{n}$.

triangle operations (y-axis) versus number of edges (x-axis)



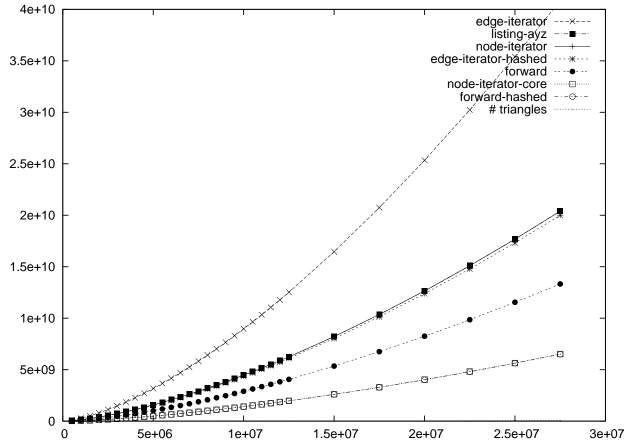
execution time in seconds (y-axis) versus number of edges (x-axis)



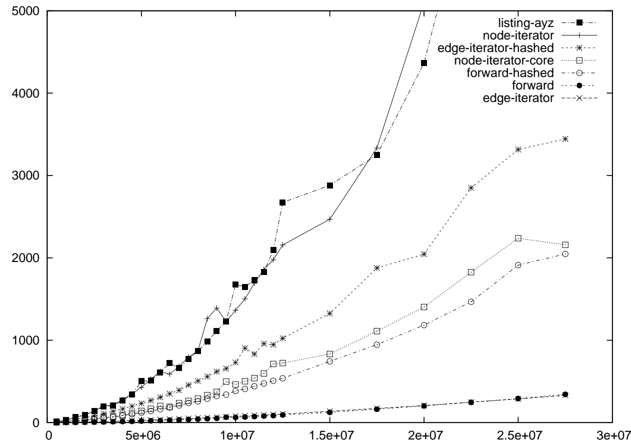
B.4 Dense $G_{n,m}$

The graphs are generated according to the $G_{n,m}$ model. The relation between n and m was chosen such that the graphs have a fixed density $m/\binom{n}{2}$ of 0.01.

triangle operations (y-axis) versus number of edges (x-axis)



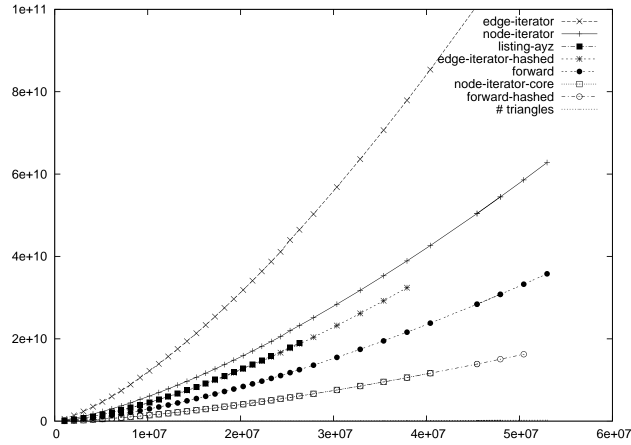
execution time in seconds (y-axis) versus number of edges (x-axis)



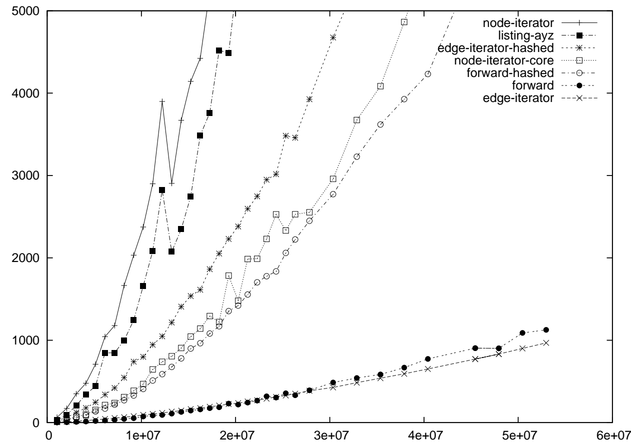
B.5 Dense G_{n,m_0} overlayed with $\Theta(\ln m_0)$ high degree nodes

The prime graphs G_{n_0,m_0} are generated like in Section B.4 and extended with parameter $h = \ln m_0$.

triangle operations (y-axis) versus number of edges (x-axis)



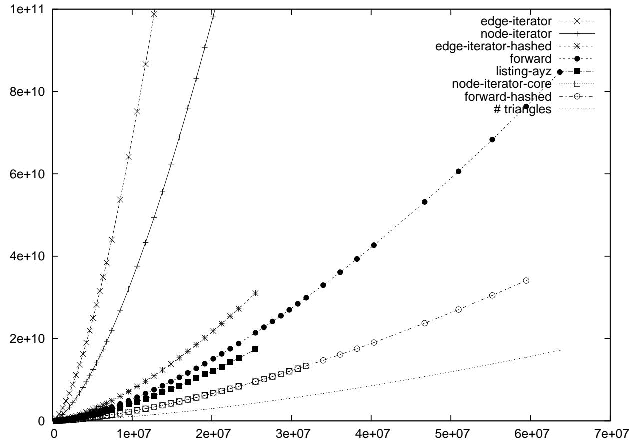
execution time in seconds (y-axis) versus number of edges (x-axis)



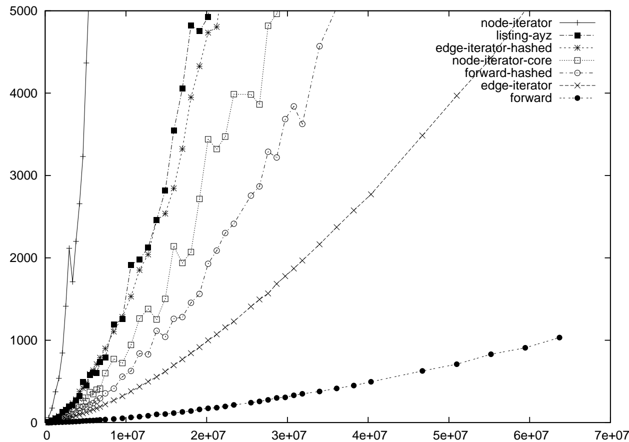
B.6 Dense G_{n,m_0} overlayed with $\Theta(\sqrt{m_0})$ high degree nodes

The same procedure as in Section B.5 is used, but here the parameter h is set to $\sqrt{m_0}$.

triangle operations (y-axis) versus number of edges (x-axis)



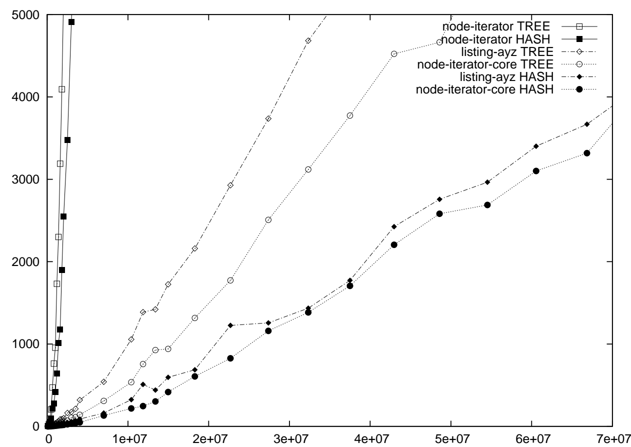
execution time in seconds (y-axis) versus number of edges (x-axis)



C Hashing versus Balanced Trees

We investigate the performance between hashed containers and balanced tree containers for storing the edges. The template from __gnu_cxx::hash_map was used for the hashed container. We store two pointers from the incident nodes of an edge. The hashing function combines two random bit fields of size size_t (one for each node) with the XOR operation. The tree container is based on the template std::set, which is an implementation of a red and black tree. An edge is coded with the lexicographic ordering of the two addresses of the incident node pointers. The results are shown in the Figure below. The graphs are constructed in the same way as the ones used in Section B.3.

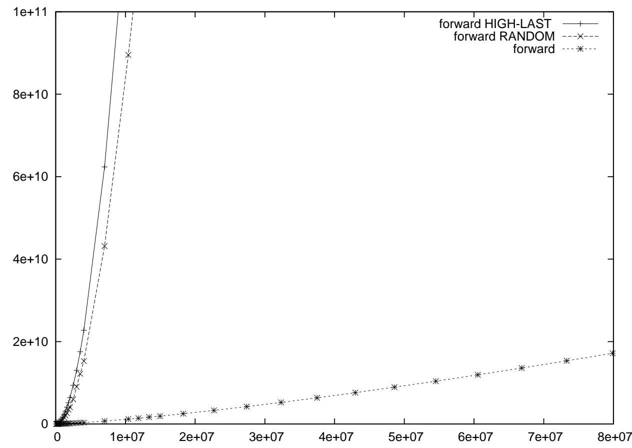
Hashing versus Balanced Trees: Execution time in seconds (y-axis) depending on the number of edges (x-axis)



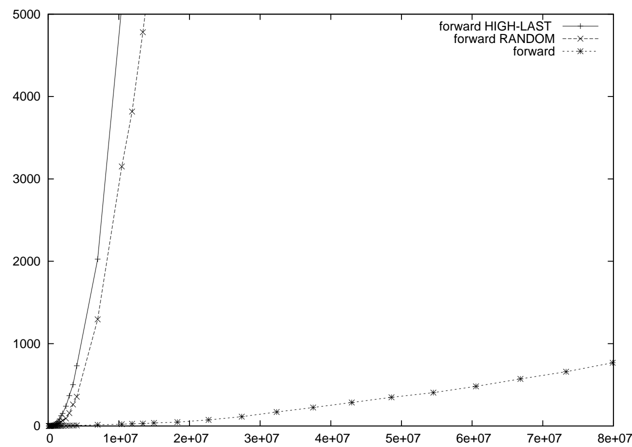
D Orderings for the *forward* Algorithm

We ordered the nodes with high degree first for the *forward* Algorithm. Here we compare the performance for the proposed ordering to the ordering with high degrees last and also to a random node ordering. The graphs are constructed in the same way as the ones used in Section B.6.

triangle operations (y-axis) versus number of edges (x-axis)



execution times in seconds (y-axis) versus number of edges (x-axis)



E Statistical Experiments of the Execution Times

The algorithms show some unsteady behavior in the experiments for the execution times. The obvious pikes for the algorithms using a hashed container for the edges are caused by rehashing of the container. But there are other influences like the variations in the structure of the generated graph or other processes running on the same machine. To investigate these influences we generated for each algorithms 100 graphs with the same procedure as in Section B.3. The sizes of the graphs are chosen such that the average execution time is not far from 60 seconds. The results are shown in the tabular below.

	execution time in seconds							
	100 graphs, one execution per graph				100 executions on one graph			
	min	mean	max	std. deviation	min	mean	max	std. deviation
node-iterator	58.83	66.94	83.18	6.15	59.48	61.92	76.96	2.80
listing-ayz	59.46	64.79	80.86	4.78	62.72	69.15	81.39	5.33
node-iterator-core	58.85	66.36	80.57	5.36	58.98	66.12	79.99	5.07
edge-iterator	59.72	60.62	61.18	4.06	59.78	60.60	61.22	0.25
edge-iterator-hashed	58.68	60.59	68.13	1.93	59.95	66.08	72.88	2.71
forward	56.11	60.08	67.49	3.18	57.74	61.15	64.31	1.66
forward-hashed	58.76	62.23	67.75	2.71	59.18	64.05	69.23	2.81