Clustering Coefficient Queries on Massive Dynamic Social Networks

Zhiyu Liu, Chen Wang, Qiong Zou, and Huayong Wang

IBM Research - China

Abstract. The Clustering Coefficient (CC) is a fundamental measure in social network analysis assessing the degree to which nodes tend to cluster together. While CC computation on static graphs is well studied, emerging applications have new requirements for online query of the "global" CC of a given subset of a graph. As social networks are widely stored in databases for easy updating and accessing, computing CC of their subset becomes a time-consuming task, especially when the network grows large and cannot fit in memory. This paper presents a novel method called "Approximate Neighborhood Index (ANI)" to significantly reduce the query latency for CC computation compared to traditional SQL based database queries. A Bloom-filter-like data structure is leveraged to construct ANI in front of a relational database. Experimental results show that the proposed approach can guarantee the correctness of a CC query while significantly reducing the query latency at a reasonable memory cost.

1 Introduction

Social network analysis is becoming a hot topic in the database and data mining communities with the fast growing popularity of social network applications in the web age. A social network is usually modeled as a graph with nodes representing people and edges representing their relationships, such as email communications, mobile calls, co-authorship, and replying to a blog. The analysis aims to understand the micro- and macro-characteristics inside the network. Based on the philosophy that friend's friends are also probably friends in a social network, Clustering Coefficient (CC) is one of the fundamental measures for assessing the degree to which nodes tend to cluster together. Traditionally, we use each node's local CC to characterize the node, and use a global CC to characterize the whole graph.

In social network analysis, CC is widely used to measure the closeness of people to a group , and the local "cliqueness", which is the stability of social communities to withstand the effect of link removal [17], e.g., co-authorship in the DBLP dataset [12]. Another useful application of CC is email and mobile messaging anti-spam. Some papers proposed social network based detection approaches to examine potential spammer's social behavior [7].

Technically, CC calculation boils down to the problem of triangle counting in graphs. It has been addressed in several approaches for exact calculation [2,4,11,10]. To eliminate the limitation of high memory consumption for

L. Chen et al. (Eds.): WAIM 2010, LNCS 6184, pp. 115–126, 2010.

large graphs, sampling methods [15,16] as well as stream or semi-stream based approximation methods have also been proposed [8,5]. However, all of the prior art focuses on a one-time calculation for a certain graph. Considering the scenario of spam detection, CC of the recipients in a spammer's distribution list tends to 0 while CC of normal users does not. For the requirement of online detection, CC will be calculated within the recipients of the sending batch in a certain time window. Because the incoming recipients are unknown at runtime, the spam detection scenario raises the new challenge of online CC queries of random subsets over a dynamic social network.

Another challenge is that massive social networks cannot be memory resident and are usually stored in relational databases, so disk accesses become the bottleneck for such kinds of query. However, for online applications, low latency of the query is always a key requirement. For example, under the spammer detection scenario, the sooner we detect the spammer the less damage it will cause. So the problem is: how to speed up the CC query on an extremely large graph that cannot fit in the memory?

This paper proposes a new indexing structure named Approximate Neighborhood Index (ANI) to solve the problem of online CC queries of a random subset of a dynamic social network. ANI is a memory resident index mapping each node to its approximate neighborhoods which is stored in a Bloom-filter like data structure. The space of ANI largely depends on the size of the Bloom filter used by each node, and grows linearly with number of nodes. For a sparse graph like a social network graph, ANI is very space-efficient. Although ANI will yield more accurate result with more space, our experiments demonstrate that two bytes per node are effective on a social network of millions of people. We also present the experimental formula to estimate a suitable Bloom-filter size given the concrete social network.

ANI can be built by initially traversing all the edges in social networks in a streaming manner, at can be incrementally updated when the social network changes dynamically. When querying the CC of a subset, ANI will return a CC upper bound and a much smaller edge set that needs to be further looked up in the database to get an accurate CC. Experimental results show a 2-10 times speedup while ensuring accuracy, and a 100+ times speedup if the CC upper bound is good enough for the application.

The remainder of this paper is organized as following: Section 2 introduces some related work, Section 3 gives the detailed design of ANI, Section 4 shows the performance of ANI through experimental results, and Section 5 concludes.

2 Related Work

2.1 Triangle Counting

CC can be calculated by counting number of triangles including a specific vertex. There has been work on exact triangle counting [2,4,11,10]. The brute-force algorithms examine each combination of nodes, resulting in a time complexity

of $O(n^3)$. The time complexity can be reduced to $O(n^{2.376})$ using matrix multiplication [9]. However, this may still be too slow to be applicable on very large graphs.

Becchetti *et al.* [5] propose two approximation algorithms for computing the local number of triangles, which are based on the idea of min-wise independent permutations. They use $O(|V|)$ space in main memory (one of the algorithms also uses $O(|E|)$ space in external memory during computation) and perform $O(log|V|)$ sequential scans over the edges in the graph.

SparsifyingEigenTriangle [15] is proposed to count triangles in power-law networks. The main idea of the algorithm is to use a low-rank approximation of the matrix which is generated via the Achlioptas-McSherry [1] sparsification of the adjacency matrix to compute the number of triangles based on the EigenTriangle algorithm [14]

Doulion [16] is an algorithm that tosses a coin to keep an edge with a certain probability p with weight equal to $\frac{1}{p}$ in order to obtain a smaller weighted graph. Then it counts each triangle as the product of the weights of the edges comprising the triangle. They prove that the expected number of triangles in G' is the number of triangles in the initial graph G, and the variance is $\frac{\Delta(p^3-p^6)+2k(p^5-p^6)}{p^6}$, where Δ is the total number of triangles in G, and k is the number of pairs of triangles that are not edge disjoint. Any triangle counting algorithm may use the idea of Doulion as a preprocessing step.

Approximate CC computation algorithms are claimed to be a hundred times faster than accurate methods with a certain accuracy using techniques such as edge sampling [8,15,16] and min-wise independent permutations [5]. However those methods are designed for computing the global CC or every node's local CC at a time.

2.2 Bloom Filter

A Bloom filter [6] is a simple space-efficient randomized data structure for representing a set in order to support membership queries. Bloom filters allow false positives but the space savings often outweigh this drawback when the probability of an error is made sufficiently low.

A standard Bloom filter is a bit array of b bits, all set to 0. There must also be k different hash functions defined, each of which maps or hashes some set element to one of the b array positions with a uniform random distribution. To add an element, feed it to each of the k hash functions to get k array positions and set the bits at all these positions to 1. To query for an element (test whether it is in the set), feed it to each of the k hash functions to get k array positions. If any of the bits at these positions are 0, the element is not in the set. If all are 1, then either the element is in the set, or the bits have been set to 1 during the insertion of other elements. A Bloom filter may yield a false positive, where it suggests that an element $x \in S$ even though it is not. Suppose that we add i items into a Bloom filter of b bits with k hash functions, Equation (1) shows the probability of false positives.

$$p = (1 - (1 - 1/b)^{ki})^k \approx (1 - e^{-ki/b})^k \tag{1}$$

Obviously, the probability of false positives decreases as b increases, and increases as i increases. For a given b and i, the value of k that minimizes the probability is $(b/i) \ln 2 \approx 0.7 \times b/i$, which gives the false positive probability of $2^{-k} \approx 0.6185^{b/i}$. Taking the optimal number of hashes, the false positive probability (when it is not greater than 0.5) can be rewritten and bounded [13]. This means that in order to maintain a fixed false positive probability, the length of a Bloom filter must grow linearly with the number of elements being filtered.

3 Proposed Method

This section first formalizes the problem, then describes the typical database solution and its shortcomings, and finally proposes a novel index built upon the database that can greatly improve the query performance.

3.1 Formalization

Let V denote a set of vertices, and E a set of edges written as $e_{i,j} \in E$ where $v_i, v_j \in V$. A directed graph is then defined as $G = (V, E)$. The neighborhood for a vertex v_i, denoted as N_i, is defined as its immediately connected neighbors as follows: $N_i = \{v_j : e_{i,j} \in E\}$. Here we give out the definition of local CC to denote how nodes in a given subset of V cluster together, named Subset Clustering Coefficient (SCC). Suppose S is a subset of V and $|S| = M$, then

$$\mathrm{SCC}(S) = \frac{|\{e_{j,k}\}|}{M \times (M-1)}, \qquad v_j, v_k \in S, e_{j,k} \in E$$

3.2 Database Only Solution

In the database-only solution, a social network is stored in a database as a table; each edge in the graph corresponds to a record (source, destination, weight) in the table. Then we create an index on both source and destination. According to the definition of SCC, we need to count the number of edges of which the source and destination are both in the given subset. Suppose the table containing the social network is called `SN_TABLE`, the following SQL query calculates the numerator of SCC for a given subset $\{v_1, v_2, \ldots, v_M\}$:

```
SELECT COUNT(*) FROM SN_TABLE WHERE SOURCE IN (V1,V2,...,VM)
AND DESTINATION IN (V1,V2,...,VM) AND SOURCE<>DESTINATION
```

The execution time of the above query depends on multiple factors, including the execution path of the SQL query, the size N of the table, the size N of the given subset, and the degrees(number of neighbours of vertex v_i in the subset) D_i of each node in the given subset. For simplicity, in the following discussion, we only consider the average degree D of the given subset. Generally speaking, the

time is mainly spent on three kinds of operations: scanning the index, fetching each node's neighbors from the table, and comparing the given subset with each node's neighbors. Table 1 shows the details. When N and M increase to a certain degree, the query latency begins to exceed the applications' restriction.

Table 1. Time complexity of CC query from database

Scan Index	Fetch Data	Compare
$O(M \times \log N)$	$O(M \times D)$	$O(DM^2)$

3.3 Approximate Neighborhood Index Using Bloom Filter

When memory is not sufficient to load in the entire graph index, and the latency of directly querying from the database is too long for a real-time application, we propose a balanced method that uses an Approximate Neighborhood Index (ANI) taking limited memory space to speed up the CC query.

As we analyze in database only solution, the query time increases with N and M. To decrease N, we can partition the database, query CC on each partition, and summarize them to get the final result; but considering the financial and management cost, it is beneficial to keep as much data as possible in each database instance. Here, we propose ANI to greatly reduce M. When the social network grows to an extremely large scale, database partitioning and ANI can be used together to control the query time.

Approximate Neighborhood Filter (ANF). For each node in the graph, we use a Bloom filter to store its neighbors called "Approximate Neighbor Filter (ANF)". Given an acceptable false positive probability p and the number of hash functions k, from Equation (1) we can get Equation (2) to estimate how many bits we need for each node to store one neighbor:

$$\frac{b}{i} = -\frac{k}{\ln(1 - p^{1/k})} \tag{2}$$

Table 2 shows the value of $\frac{b}{i}$ of some typical k, p combinations.

Table 2. b/i of typical k, p combinations

$k, p \rightarrow b/i$	0.05	0.01	0.001	0.0001	0.00001
3	6.5	12.4	28.5	63.1	137.7
4	6.2	10.5	20.4	38.0	69.1
5	6.3	9.8	17.3	29.0	47.5

If the false positive requirement is not that strict (say 1% is acceptable), the Bloom-filter approach needs about one to two bytes to store one edge. Answering

whether a node is the neighbor of the others requires $O(1)$ time, thus reducing the time complexity of CC to $O(N)$. However, the shortcomings of using Bloom filters are also obvious. First, since one position in the Bloom filter may be set to "1" multiple times by different items, deleting an item becomes complex. Simply setting the correspondence positions to "0" may result in false negatives. One solution is to add a counter to each position; another solution is to rebuild a node's Bloom filter with the rest of its neighbors, which requires that accurate neighbors' information need to be stored somewhere. Second, the "weight" of an edge cannot be stored in a Bloom filter, so this method cannot estimate weighted CC. Third, the existence of false positives makes it only possible to estimate an upper bound of the accurate CC.

A social network is known to have the "long tail" property [3,17], which means it is a sparse graph. For subset communities containing hundreds of nodes, the CC may be as low as 10^{-4} or even 10^{-5} in practice. In order to distinguish between a spammer, whose contacts' CC is almost always 0 in any time intervals, and a normal user, the acceptable false positive probability of CC is 10^{-6}. Suppose the subset contains M nodes $v_1, v_2, \ldots, v_M$, for each node v_i, its neighbor filter's false positive is p_i, we can get the false positive of their CC by Equation (3):

$$\mathrm{FP}(CC) = \frac{(M-1)(p_1 + p_2 + \cdots + p_M)}{M(M-1)} = (p_1 + p_2 + \cdots + p_M)/M \quad (3)$$

So it requires the average false positive probability of the ANF to be less than 10^{-6}, which makes it no longer space effective to use the Bloom-filter only solution.

Furthermore, the social network is evolving with time, so deleting some nodes or edges cannot be avoided, which means the Bloom filter must be used with another data store to keep an accurate record of the social graph.

So, we propose a Bloom-filter based index called "ANI" in front of the relational database to guarantee the correctness for the CC query while significantly reducing the query latency at a reasonable memory cost.

ANI Construction. We first create a hash map for all the nodes. It maps each node to its own ANF. Then we traverse the graph in the database in a streaming way. Each time we encounter a (source, destination) pair, we insert the destination into the source's ANF. For simplicity, we use a fixed length of Bloom filters for all of the nodes. As different nodes have different degrees, the false positives will vary according to degrees. However, in a social network, most nodes have small degrees, so as long as the length of Bloom filter can control the false positives of the majority of nodes, the index will be very effective. We show the piece of code that builds the index from the database in Algorithm 1.

Query Through ANI. For a given subset $S = \{v_1, v_2, \ldots, v_M\}$, we can look up the ANI, using each node's ANF to filter out the edges that definitely do not exist and to get a set of edges that could exist in the subset. Specifically, for each

```
Input: SN_TABLE
Output:ANI
hstmt ← SQLExecDirect(SELECT * FROM SN_TABLE);
sr ← SQLFetch(hstmt);
while sr is SUCCESS do
    src ← SQLGetData(hstmt, 1, SQL_INTEGER);
    dest ← SQLGetData(hstmt, 2, SQL_INTEGER);
    itm ← ANI.Find(src);
    if src is not found then
        neighborfilter ← BMFilter();
        ANI.Insert(src, neighborfilter);
    else
        neighborfilter ← itm.neighborfilter;
    end
    neighborfilter.Add(GetMask(dest));
end
```

Algorithm 1. ANI construction

$v_i \in S$, we will first look up its ANF from the ANI, and then test whether v_j ($v_j \in S$ and $i \neq j$) is in v_i's ANF. The total size of the result set is $e + \sum_1^M p_i * d_i$, where p_i, d_i is the false positive probability and the degree of v_i, and e is the real edge number. Then we can further remove false positives using the database to get an accurate CC value. The query can be rewritten as:

```
SELECT COUNT (*) FROM SN_TABLE
  WHERE SOURCE=V1, DESTINATION IN RemainingNeighborListOfV1
     OR SOURCE=V2, DESTINATION IN RemainingNeighborListOfV2
      ...
     OR SOURCE=VP, DESTINATION IN RemainingNeighborListOfVP
```

Algorithm 2 shows the pseudo-code that transfers a subset "group" into a small graph that filters out most non-existent edges by looking up neighbor indices.

ANI Update. When the social network evolves with time, if a node or an edge is inserted into the graph, the ANF will also insert the new node or edge. If a node or edge is deleted from the graph, only the database is updated accordingly. Obviously, the false positives may increase as the social network evolves. We solve this problem by periodically rebuilding the neighbor index from the database.

Efficiency Analysis of ANI. By querying the database with hints from the ANI, we can reduce the time of index scanning, data fetching, and neighbor list comparing. In a sparse graph like a social network, one node may only connect to a few or even zero other nodes within a subset. So some nodes can be safely removed from the source list. For each source, its neighbors can also be further restricted to a relatively small area. Roughly speaking, up to $(1-CC-p)$ of query workload can be reduced. Here we can see that the efficiency of ANI is mainly

```
Input: group, ANI
Output: smallgraph, CCupperbound
for Each node n1 in group do
    neighborfilter =ANI.find(GetMask(n1).neighborfilter);
    for Each node n2 in group do
        if n2 == n1 then
            continue;
        end
        if n2 exists in neighborfilter then
            neighborlist.add(n2);
            CCupperbound ++ ;
            need = true;
        end
    end
    if need then
        smallgraph.insert(n1,neighborlist);
    end
end
```

Algorithm 2. Query through ANI

influenced by the average false positive probability p and the real CC value of the subset. Since CC is very small in sparse graphs and p can be controlled to a small value by proper ANF, the reduced workload is obvious.

4 Experiments

This section evaluates the performance of the neighbor index by measuring the percentage time reduction from using ANI in front of a database. The data set we used containing 3,091,943 users and 12,108,369 "knowing" relations between them. The data set is generated from one week's server log of a short messaging center of a city by removing private information. One person is considered to "know" another person if he/she frequently contacts him/her. Fig. 1 shows the cumulative edge degree distribution of the data set. We can see that about 80% of the nodes have a degree less than 4, and over 90% nodes have a degree less than 8.

We use two kinds of CC query series. The fist one is called "fixed size same degree" (FSSD) which contains 27 queries generated by the program. Each query in the FSSD series queries CC of a subset of 1,000 elements with the same degree. The degree varies from 1 to 27. The second query series called "real trace" (RT) is a replay of the real queries observed in the real application. If a user contacts over 10 different people in the past 10 minutes, a query of CC of his previously contactors is triggered. The RT series contains 142,848 queries and the subset size varies from 10 to 600.

In the FSSD series experiments, we mainly demonstrate how false positive influences the performance of ANI. All the queries in the FSSD series have very

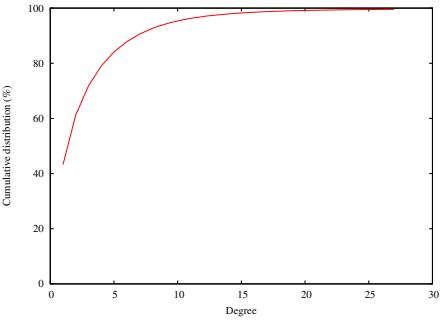

Fig. 1. Cumulative edge degree distribution of the dataset

small CC (10^{-4} to 10^{-5}) because of two reasons: first, the size of the subset is very large; and second, the elements are all randomly selected among all nodes that have a certain degree. So the main factor that differentiates the performance of ANI in different queries is the probability of false positives. Since we use a fixed-size Bloom filter for all nodes' ANF, a node's false positive probability will increase nearly exponentially with its degree. False positives are also influenced by the size of the Bloom filter, so we choose two different sizes of Bloom filter to build the ANF: 32-bit and 64-bit. With the 32-bit Bloom filter, nodes with degree no more than 4 can have a false positive probability below 0.05, and with the 64-bit Bloom filter, nodes with degree no more than 8 can have a false positive probability below 0.05. Fig. 2 shows that the 64-bit Bloom filter has better performance than the 32-bit Bloom filter, and the performance decreases as the average degree increases, as predicted by our analysis. When the average degree is less than 5, over 90% of time can be saved by using ANI of both sizes. When the degree is larger than 5, the performance of the 32-bit Bloom filter decreases sharply. But the 64-bit Bloom filter can save 70% to 99% query time in all of the queries.

Fig. 2 also plots the upper bound of CC returned when looking up ANI, which equals $(1+p) \times \mathrm{CC}$. Consistently with our analysis, the performance curves have an opposite trend than the upper bound factor. Comparing `CC_upper_bound_64`, `CC_upper_bound_32`, and CC, we can see that the space between the upper bound and real value greatly increases as the degree increases. So in order to get an efficient ANI, we need to carefully choose the size of the Bloom filter. Our suggestion is that the size be big enough to ensure that at least 90% of nodes' ANF have a false positive probability less than 5%, thus the average false positive probability of CC can be controlled under $90\% \times 5\% + 10\% \times 1 = 15\%$.

The experiments on the RT series mainly show the effectiveness of ANI in real applications. First, we analyze how much the workload can be reduced by ANI using a 64-bit Bloom filter. In Fig. 3, 142,848 queries are grouped and ordered according to their CC on the x-axis; three curves represent the percentage of removed edges, percentage of removed sources, and cumulative distribution of query numbers. For the queries that have the same CC, we plot their average

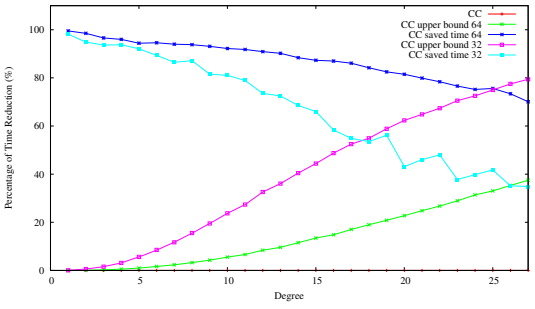

Fig. 2. Performance evaluation of FSSD series

value on the y-axis. We can see that in about 40% CC queries, we can remove over 40% source nodes that don't connect to any other node in the given subset. And totally in about 80% queries, 80% edges can be filtered out by ANI.

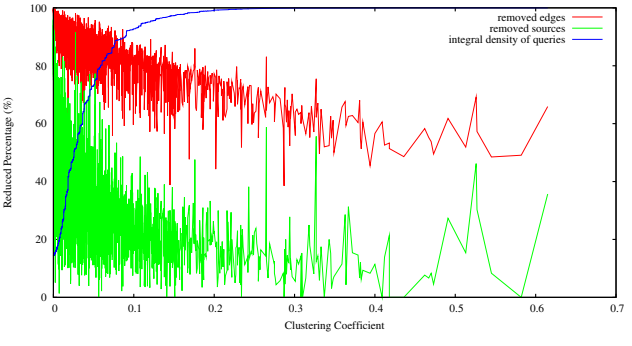

Fig. 3. Workload analysis of RT series

Comparing CC and `CC_upper_bound` in Fig. 4, we can see that most false positives probabilities are less than 20%, and the average false positive probability is 8.77% . The saved time also shows the same trend as we see in the FSSD experiments: it goes in the opposite direction of `CC_upper_bound`. By using a neighbor index with a 64-bit Bloom filter, we can save 67.25% time on average. In some rare case, the false positive probability is nearly 0.5. The reason is that there are many supernodes (having very high degree) in the subset, so the false positives of such queries are much higher than the average level. In some other rare cases, using a neighbor index doesn't save any time or even costs more time. We call such cases "time-out" cases. The overhead of looking up the index is a reason but not the main reason, because the neighbor index is small enough to be loaded into main memory and looking up the index costs no more than several milliseconds, which is a very small part of the whole query time. And in

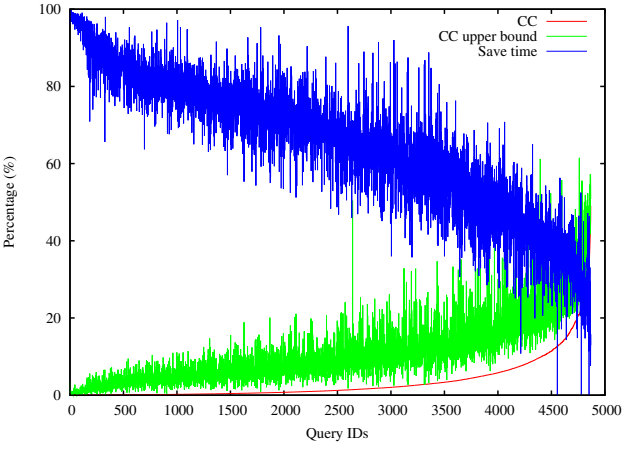

Fig. 4. Performance evaluation of RT series

each repeated run, we have about 1% "time-out" cases, but none of them are the same. One possible explanation is that the unusual latency is caused by random factors in the system, like a buffer flush just happened.

Finally, we measure the time to build the index. The average time to build an index for the dataset in the above tests is 48.1s. The time is mainly spent on reading data from the database, so it will be linearly increased with the size of the data set. And in practical applications, rebuilding the index daily or weekly is enough for a social network.

5 Conclusion

In this paper, we present a Bloom-filter based index "ANI" built upon a relational database to improve the performance of the subset CC query on extreme-scale social networks. We can get 2-10 times speedup while ensuring the accuracy and 100+ times speedup if a certain false positive is acceptable. We also present the experimental formula to estimate a suitable Bloom-filter size given the concrete social network. Although the design of ANI is triggered by the CC query, it could further be used in many other graph queries like sub-graph matching and reachability queries.

References

1. Achlioptas, D., Mcsherry, F.: Fast computation of low-rank matrix approximations. J. ACM 54(2), 9 (2007)
2. Alon, N., Yuster, R., Zwick, U.: Finding and counting given length cycles. Algorithmica 17(3), 209–223 (1997)
3. Anderson, C.: The Long Tail: Why the Future of Business Is Selling Less of More. Hyperion, vol. 12 (2004)

4. Batagelj, V., Mrvar, A.: A subquadratic triad census algorithm for large sparse networks with small maximum degree. Social Networks 23, 237–243 (2001)
5. Becchetti, L., Boldi, P., Castillo, C., Gionis, A.: Efficient semi-streaming algorithms for local triangle counting in massive graphs. In: KDD 2008: Proceeding of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining, pp. 16–24. ACM, New York (2008)
6. Bloom, B.H.: Space/time trade-offs in hash coding with allowable errors. Commun. ACM 13(7), 422–426 (1970)
7. Oscar Boykin, P., Roychowdhury, V.P.: Leveraging social networks to fight spam, vol. 38, pp. 61–68. IEEE Computer Society Press, Los Alamitos (2005)
8. Buriol, L.S., Frahling, G., Leonardi, S., Marchetti-Spaccamela, A., Sohler, C.: Counting triangles in data streams. In: PODS 2006: Proceedings of the twenty-fifth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems, pp. 253–262. ACM, New York (2006)
9. Coppersmith, D., Winograd, S.: Matrix multiplication via arithmetic progressions. In: STOC 1987: Proceedings of the nineteenth annual ACM symposium on Theory of computing, pp. 1–6. ACM, New York (1987)
10. Coppersmith, D., Winograd, S.: Matrix multiplication via arithmetic progressions. Journal of Symbolic Computation 9(3), 251–280 (1990)
11. Itai, A., Rodeh, M.: Finding a minimum circuit in a graph. SIAM Journal on Computing 7(4), 413–423 (1978)
12. Ley, M.: Dblp bibliography
13. Starobinski, D., Trachtenberg, A., Agarwal, S.: Efficient pda synchronization. IEEE Transactions on Mobile Computing 2(1), 40–51 (2003)
14. Tsourakakis, C.E.: Fast counting of triangles in large real networks without counting: Algorithms and laws. In: ICDM 2008: Proceedings of the 2008 Eighth IEEE International Conference on Data Mining, pp. 608–617. IEEE Computer Society, Washington (2008)
15. Tsourakakis, C.E., Drineas, P., Michelakis, E., Koutis, I., Faloutsos, C.: Spectral counting of triangles in power-law networks via element-wise sparsification. In: ASONAM, pp. 66–71 (2009)
16. Tsourakakis, C.E., Kang, U., Miller, G.L., Faloutsos, C.: Doulion: counting triangles in massive graphs with a coin. In: KDD 2009: Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining, pp. 837–846. ACM, New York (2009)
17. Watts, D.J., Strogatz, S.H.: Collective dynamics of'small-world'networks. Nature 393(6684), 409–410 (1998)