Practical algorithms for triangle computations in very large (sparse (power-law)) graphs

Matthieu Latapy *

Abstract

Finding, counting and/or listing triangles in large graphs are natural problems, which received recently much attention because of their importance in complex network analysis. However, the time and/or space requirements of known algorithms limit our ability to solve these problems in practice. We give here a quick overview of previous results, with a special emphasis on space requirements, which were not considered before. We then detail the analysis a recently proposed algorithm which surpasses all previous ones. We propose an improvement of this algorithm that significantly reduces its space requirements, as well as a new algorithm with similar performances. These two algorithms have the additional advantage of performing better on power-law graphs, which we also study. This explains their high efficiency observed in practice. They both make it possible to practically solve triangle problems in cases that were previously out of reach, which we illustrate with a typical example.

1 Introduction.

A *triangle* in an undirected graph is a set of three vertices such that each possible edge between them is present in the graph. The problems of deciding if a given graph contains a triangle, counting the number of triangles in the graph, and listing all of them are called *finding*, *counting* and *listing* respectively.

These problems may be considered as natural and fundamental algorithmic questions, and have been studied as such [18, 10, 2, 3, 25, 26]. Moreover, they gained recently much practical importance because they are central in *complex network analysis* [9, 1, 14]: triangles are the basis for two important complex network statistics, namely the clustering coefficient [28, 24] and the transitivity ratio [16, 15, 24]. In this context, they also play a key role in the study of motif occurrences [22, 29].

In complex network studies, one often deals with huge graphs, typically with several million vertices/edges and up to a few billions. Both the time and space requirements of triangle computations then are key issues, and space often is a limitation in their practical tractability. Moreover, space complexity raises interesting theoretical questions. Despite this, previous authors focused on time complexity only.

In this paper, we propose compact algorithms that make it possible to solve triangle problems in cases that were previously out of reach because of space requirements of fast algorithms. Moreover, we prove that they perform even better on graphs with power-law degree distribution, which is a propertiy shared by most real-world complex networks.

*LIP6, CNRS and Université Pierre et Marie Curie (Paris 6), 4 place Jussieu, 75005 Paris, France. Firstname.Lastname@lip6.fr

2 Notations.

We consider an undirected graph $G = (V, E)$ with $n = |V|$ vertices and $m = |E|$ edges. We suppose that G is simple ($(v, v) \notin E$ for all v, and there is no multiple edge). We denote by $N(v) = \{u \in V,\ (v, u) \in E\}$ the neighborhood of $v \in V$ and by $d(v) = |N(v)|$ its degree. We also denote by $d_{\max}$ the maximal degree in G.

In the context of complex network studies, the difference between an algorithm with a given time complexity and an algorithm twice as fast generally is not crucial. Space limitations are much stronger and dividing space complexity by a constant is a significant improvement (it often makes the difference between tractable and untractable computations in practice) [1]. We will give an illustration of this in Section 7.

In order to capture this, we will use a notation in addition to the usual $O()$ and $\Theta()$ ones. We will say that a space complexity is in $\Theta'(f(n, m))$ if the cost of the algorithm is exactly $\sigma \cdot f(n, m) + c$ where c is any constant and σ is the space needed to encode a vertex, an integer between 0 and n, or a pointer. Though it actually is in $\Theta(\log(n))$, we will follow the classical convention assuming that σ is a constant; taking this into account would make the text unclear, and would bring little information, if any.

With this notation, the adjacency matrix of G needs $\Theta'(\frac{n^2}{\sigma}) \subseteq \Theta(n^2)$ space, because the matrix needs n^2 bits (and an integer and a pointer). An adjacency list representation of G (array of n linked lists) needs $\Theta'(4m + n) \subseteq \Theta(m)$ space (a vertex and a pointer for each edge in both directions plus n pointers), and an adjacency array representation (array of n arrays) needs only $\Theta'(2m + n) \subseteq \Theta(m)$ space. This is why this last representation generally is preferred when dealing with huge graphs. We will use it in the following, but results on such representations may easily be converted into results on adjacency list representations (only the space complexity in terms of $\Theta'()$ is affected).

Each adjacency array in an adjacency array representation of G may moreover be sorted. This can be done in place in $\Theta(m \cdot \log(n))$ time and $\Theta'(2m+n) \subseteq \Theta(m)$ space (only a constant space is needed in addition to the representation of G).

Finally, notice that, in several cases, we will not give the space needs in terms of $\Theta'()$ because the algorithm complexity is prohibitive; the precise space requirements then are of little interest, and they would make the text intricate. We will enter in these details only in cases where the time and space complexities are small enough to make the precise space cost interesting.

3 State of the art.

We present here a quick overview of previous results on triangle problems (detailed and unified presentation of these algorithms is given in [20]). In most cases, space requirements were not discussed in the original papers, but their analysis follows easily from the algorithm descriptions.

Fast *finding* and *counting* algorithms.

The fastest algorithm known for *finding* and *counting* triangles relies on fast matrix product [18, 2, 3, 12] and has an $O(n^\omega)$ time complexity, where $\omega < 2.376$ is the fast matrix product

[1] To this regard, approaches not requiring the graph to be stored in central memory like streaming algorithms [17, 4, 19], or compressed graph representations [5, 6], have been developed. This is however out of the scope of this paper.

exponent [12]. This approach however leads to a $\Theta(n^2)$ space complexity. Notice moreover that it does not solve triangle *listing*.

One can design faster algorithms if G is sparse (*i.e.* m is significantly lower than its maximal value: $m \in o(n^2)$). In [18], it was first proved that triangle *finding* and *counting* can be solved in $\Theta(m^{\frac{3}{2}})$ time, $\Theta(n^2)$ space. This result has been improved on special classes of graphs in [10]. A significant progress has been made in [3, 2], where a *finding* and *counting* algorithm in $O(m^{\frac{2\cdot\omega}{\omega+1}}) \subset O(m^{1.41})$ time and $\Theta(n^2)$ space was proposed.

All these algorithms are asymptotically very fast, but they have a prohibitive space cost: they rely on the use of the adjacency matrix of the graph, and moreover in most cases the computation produces intermediary matrices which may not be sparse even if the adjacency matrix is. Therefore, their space complexity is in $\Theta(n^2)$. This is why one generally uses a (more compact) *listing* algorithm to solve *finding* and *counting* in practice.

Basic listing algorithms.

First notice that there may be $\Theta(n^3)$, or $\Theta(m^{\frac{3}{2}})$, triangles in G. This gives lower bounds for the time complexity of any listing algorithm.

One may trivially reach the $\Theta(n^3)$ bound with the matrix representation of G by testing in $\Theta(1)$ time any possible triple of vertices [25, 26]. This has a $\Theta(n^2)$ space complexity because of the matrix, but if G is very dense, almost all triple of vertices forms a triangle and no better performances can be attained.

However, in practice G often is sparse, and then algorithm complexities in terms of m are more appealing. Two simple algorithms introduced in [18], *vertex-iterator* and *edge-iterator*, are widely used in this context.

The first one computes, for each vertex v, the number of triangles containing it, *i.e.* $|N(v) \times N(v) \cap E|$. It runs in $\Theta(m \cdot d_{\max})$ time, which is very efficient when $d_{\max}$ is bounded by a constant. However, it has the serious drawback of needing both the adjacency matrix and an adjacency array representation of the graph in input, leading to a $\Theta(n^2)$ space complexity.

Algorithm *edge-iterator* computes for each edge (u, v) the number of triangles containing it, *i.e.* $|N(u) \cap N(v)|$. Its time complexity is the same as *vertex-iterator* if the representation is sorted [18, 25, 26], and $\Theta(m(d_{\max} + \log(n)))$ otherwise, but it uses an adjacency array representation and thus has a $\Theta'(2m + n) \subseteq \Theta(m)$ space complexity (it needs only a constant amount of space in addition to the graph representation).

Another $O(m \cdot n)$ time algorithm was proposed in [23] for a more general problem. In the case of triangles, however, it does not improve *vertex-iterator* and *edge-iterator*, which are simpler.

Notice that none of these algorithm reaches the $\Theta(m^{\frac{3}{2}})$ time complexity bound, which is particularily appealing for sparse graphs.

Time-optimal listing algorithms for sparse graphs.

The $\Theta(m^{\frac{3}{2}})$ time *finding* algorithm proposed in [18] may be extended easily to obtain a *listing* algorithm with the same complexity. Likewise, as proposed in [25, 26], it is also easy to modifiy the fast *finding* and *counting* algorithm proposed in [3, 2] to obtain a *listing* algorithm in $\Theta(m^{\frac{3}{2}})$ time.

However, these algorithms both need the adjacency matrix of G, and thus have a $\Theta(n^2)$ space cost, reducing significantly their practical interest.

In [25, 26], the authors propose Algorithm 1 (*forward*), and show that it lists all the triangles in a graph in $\Theta(m^{\frac{3}{2}})$ time and $O(m)$ space only, which is significantly better than all previous

algorithms for sparse graphs. In addition, they present an extensive experimental evaluation which shows that *forward* is very efficient in practice and surpasses all previous algorithms on real-world complex networks.

Algorithm 1 – *forward.* *Lists all the triangles in a graph [25, 26].*

Input: an adjacency array representation of G
1. number the vertices with an injective function $\eta()$
   such that $d(u) > d(v)$ implies $\eta(u) < \eta(v)$ for all u and v
2. let A be an array of n arrays initially empty
3. for each vertex v taken in increasing order of $\eta()$:
   3a. for each $u \in N(v)$ with $\eta(u) > \eta(v)$:
      3aa. for each w in $A[u] \cap A[v]$: output triangle $\{u, v, w\}$
      3ab. add v to $A[u]$

4 Analysis and improvement of the *forward* algorithm.

This section is devoted to a new proof of the time and space complexities of *forward*, which will then be useful in reducing its space needs.

Theorem 1 *Given an adjacency array representation of G, Algorithm 1 (*forward*) lists all its triangles in $\Theta(m^{\frac{3}{2}})$ time and $\Theta'(3m + 3n) \subseteq \Theta(m)$ space.*

Proof: For each vertex x, let us denote by $A(x)$ the set $\{y \in N(x),\ \eta(y) < \eta(x)\}$; this set contains only neighbors of x with degree larger than or equal to the one of x itself. For any triangle $t = \{a, b, c\}$ one can suppose without loss of generality that $\eta(c) < \eta(b) < \eta(a)$. One may then discover t by discovering that c is in $A(a) \cap A(b)$.

This is what the algorithm does. To show this it suffices to show that $A[u] \cap A[v] = A(u) \cap A(v)$ when computed in line 3aa.

First notice that when one enters in the main loop (line 3), the set $A[v]$ contains all the vertices in $A(v)$. Indeed, u was previously treated by the main loop since $\eta(u) < \eta(v)$, and, during this, lines 3 and 3ab ensure that it has been added to $A[v]$ (just replace u by v and v by u in the pseudocode). Moreover, $A[v]$ contains no other element, and thus it is exactly $A(v)$ when one enters the main loop.

When entering the main loop for v, $A[u]$ is not equal to $A(u)$ but it contains all the vertices w in $A(u)$ such that $\eta(w) < \eta(v)$. Therefore, the intersections are equal: $A[u] \cap A[v] = A(u) \cap A(v)$, and thus the algorithm is correct.

Let us turn to complexity analysis. First notice that line 1 can be achieved in $\Theta(n \cdot \log(n))$ time and $\Theta'(n)$ space.

Now, note that lines 3 and 3a are nothing but a loop over all edges, thus in $\Theta(m)$. Inside the loop, the expensive operation is the intersection computation. To obtain the claimed complexity, it suffices to show that both $A[u]$ and $A[v]$ contain $O(\sqrt{m})$ vertices: since each structure $A[x]$ is trivially sorted by construction, this is sufficient to ensure that the intersection computation is in $O(\sqrt{m})$.

For any vertex x, by definition of $A(x)$ and $\eta()$, $A(x)$ is included in the set of neighbors of x with degree at least $d(x)$. Suppose x has $\omega(\sqrt{m})$ such neighbors: $|A(x)| \in \omega(\sqrt{m})$. But all these vertices have degree at least equal to the one of x, with $d(x) \geq |A(x)|$, and thus they have all together $\omega(m)$ edges, which is impossible. Therefore one must have $|A(x)| \in O(\sqrt{m})$,

and since $A[x] \subseteq A(x)$ this proves the $O(m^{\frac{3}{2}})$ time complexity. This bound is tight since the graph may contain $\Theta(m^{\frac{3}{2}})$ triangles.

The space complexity is obtained when one notices that each edge induces the storage of exactly one vertex (line 3ab), leading to a space requirement in $\Theta'(3m + 3n)$: $\Theta'(2m + n)$ for the graph representation plus $\Theta'(m + n)$ for A and $\Theta'(n)$ for η. □

Thanks to this proof, it is now easy to modify *forward* in order to reduce significantly its space needs. This leads to the following result.

Algorithm 2 – *compact-forward.* *Lists all the triangles in a graph.*

Input: an adjacency array representation of G

1. number the vertices with an injective function $\eta()$
   such that $d(u) > d(v)$ implies $\eta(u) < \eta(v)$ for all u and v
2. sort the simple compact representation according to $\eta()$
3. for each vertex v taken in increasing order of $\eta()$:
  3a. for each $u \in N(v)$ with $\eta(u) > \eta(v)$:
    3aa. let u' be the first neighbor of u, and v' the one of v
    3ab. while there remain untreated neighbors of u and v and $\eta(u') < \eta(v)$ and $\eta(v') < \eta(v)$:
      3aba. if $\eta(u') < \eta(v')$ then set u' to the next neighbor of u
      3abb. else if $\eta(u') > \eta(v')$ then set v' to the next neighbor of v
      3abc. else:
        3abca. output triangle $\{u, v, u'\}$
        3abcb. set u' to the next neighbor of u
        3abcc. set v' to the next neighbor of v

Theorem 2 *Given an adjacency array representation of G, Algorithm 2 (*compact-forward*) lists all its triangles in $\Theta(m^{\frac{3}{2}})$ time and $\Theta'(2m + 2n) \subseteq \Theta(m)$ space.*

Proof: Recall that, as explained in the proof of Theorem 1, when one computes the intersection of $A[v]$ and $A[u]$ (line 3aa of Algorithm 1 (*forward*)), $A[v]$ is the set of neighbors of v with number lower than $\eta(v)$, and $A[u]$ is the set of neighbors of u with number lower than $\eta(v)$. If the adjacency structures encoding the neighborhoods are sorted according to $\eta()$, we then have that $A[v]$ is nothing but the beginning of $N(v)$, truncated when we reach a vertex v' with $\eta(v') > \eta(v)$. Likewise, $A[u]$ is $N(u)$ truncated at u' such that $\eta(u') > \eta(v)$.

Algorithm 2 (*compact-forward*) uses this: lines 3ab to 3abcc are nothing but the computation of the intersection of $A[v]$ and $A[u]$, which are supposed to be stored at the beginning of the adjacency structures, which is done in line 2. All this has no impact on the asymptotic time cost, and the A structure does not have to be explicitly stored.

Notice now that line 1 has a $O(n \cdot \log(n))$ time and $\Theta'(n)$ space cost. Moreover, sorting the simple compact representation of G (line 2) is in $O(m \cdot \log(n))$ time and $\Theta(1)$ space. These time complexities play no role in the overall complexity, but the space complexities induce a $\Theta'(n)$ additional space cost for the overall algorithm. □

Compared to previous algorithms, *forward* and *compact-forward* have several key advantages. The main one is that they do not need the adjacency matrix of the graph, thus avoiding the $\Theta(n^2)$ space complexity. Their space complexities are very tight, and they are the first time-optimal algorithms on sparse graphs with such space requirements. Moreover, both are

very easy to implement, which also has a role in their practical efficiency demonstrated in [25, 26].

Note moreover that one does not need to store the whole adjacency arrays representing G in order to list the triangles using Algorithm 2 (*compact-forward*): if the adjacency array of each vertex v contains only its neighbors u such that $\eta(u) > \eta(v)$ then the algorithm still works. We obtain the following result.

Corollary 3 *If the input adjacency arrays representing G are already sorted according to $\eta()$, then it is possible to list all the triangles in G in time $\Theta(m^{\frac{3}{2}})$ and space $\Theta'(m+n) \subseteq \Theta(m)$.*

This last method is very compact (it does not even need to store the whole graph), and moreover all the preprocessing needed to reach these performances (computing the degree of each vertex, sorting them according to their degree, translating the adjacency arrays, and sorting them) can be done in $\Theta(m \cdot \log(n))$ time and $\Theta'(2n) \subseteq \Theta(n)$ space only. In cases where the available memory is too limited to store the whole graph, this makes this method very appealing.

5 A new algorithm.

Performances of the algorithms discussed above basically rely on the fact that they avoid considering each pair of neighbors of high degree vertices, which would have a prohibitive cost. They do so by managing low degree vertices first, which has the consequence that most edges involved in the highest degrees have already been treated when the algorithm comes to these vertices. Here we take a quite different approach.

First notice that one may compute all the triangles containing a given vertex v as follows. Construct an array A of n bits and set them to 0. Now, for each neighbor u of v set $A[u]$ to 1. Notice that A is nothing but the v-th line of the adjacency matrix. Finally, for each neighbor u of v, consider each neighbor w of u and check if $A[w]$ is equal to 1. If it is, then it means that the w is also linked to v, and thus we have found a triangle. Conversely, every triangle containing v may be discovered this way. This leads to the following results.

Lemma 4 *Given an adjacency array representation of G, it is posible to list all the triangles in G containing a given vertex v in $\Theta(m)$ time and $\Theta'(\frac{n}{\sigma}) \subseteq \Theta(n)$ space.*

Algorithm 3 – *new-listing.* *Lists all the triangles in a graph.*

Input: a sorted adjacency array representation of G, and an integer K

1. for each vertex v in V:
  1a. if $d(v) > K$ then, using the method of Lemma 4:
    1aa. output all triangles $\{v, u, w\}$ such that $d(u) > K$, $d(w) > K$ and $v > u > w$
    1ab. output all triangles $\{v, u, w\}$ such that $d(u) > K$, $d(w) \leq K$ and $v > u$
    1ac. output all triangles $\{v, u, w\}$ such that $d(u) \leq K$, $d(w) > K$ and $v > w$
2. for each edge (v, u) in E:
  2a. if $d(v) \leq K$ and $d(u) \leq K$ then:
    2aa. if $u < v$ then output all triangles containing (u, v) by computing $N(u) \cap N(v)$

Theorem 5 *Given a sorted adjacency array representation of G, Algorithm 3 (*new-listing*) lists all its triangles in $\Theta(m^{\frac{3}{2}})$ time and $\Theta'(2m+n+\frac{n}{\sigma}) \subseteq \Theta(m)$ space if one takes $K \in \Theta(\sqrt{m})$.*

Proof: Let us first study the complexity of the algorithm as a function of K. For each vertex v with $d(v) > K$, one lists the number of triangles containing v in $\Theta(m)$ time and $\Theta'(\frac{n}{\sigma}) \subseteq \Theta(n)$ space (Lemma 4) (the conditions in lines 1aa to 1ac, as well as the one in line 2aa, only serve to ensure that each triangle is listed exactly once). Then, one lists the triangles containing edges whose extremities are of degree at most K; this is done by line 2aa in $\Theta(K)$ time and $\Theta(1)$ space for each edge, thus a total in $O(m \cdot K)$ time and $\Theta(1)$ space.

Finally, the space needs of the whole algorithm are independent of K and it is only in $\Theta'(\frac{n}{\sigma}) \subseteq \Theta(n)$ in addition to the $\Theta'(2m + n) \subseteq \Theta(m)$ space representation of G. Its time complexity is in $O(\frac{m}{K} \cdot m + m \cdot K)$ time, since there are $O(\frac{m}{K})$ vertices with degree larger than K. In order to minimize this, we take K in $\Theta(\sqrt{m})$, which leads to the announced time complexity. □

Theorems 2 and 5 improve previous results since they show that the $\Theta(m^{\frac{3}{2}})$ time complexity may be achieved in only $\Theta(n)$ space in addition to the $\Theta(m)$ space needed to store G. We will see in Section 7 that this makes a significant difference in practice.

Note however that it is still unknown whether there exist algorithms with this time complexity with only $o(n)$ space requirements in addition to the $\Theta(m)$ graph representation. We saw that *edge-iterator* achieves such space requirements, but it needs $\Theta(m \cdot d_{\max}) \subseteq O(m \cdot n)$ time.

6 The case of power-law graphs.

Several results we have presented until now take advantage of the fact that most large graphs met in practice are sparse; designing algorithms with complexities expressed in term of m rather than n then leads to significant improvements.

Going further, it has been observed since several years that most large graphs met in practice have another important characteristic in common: their degrees are very heterogeneous. More precisely, in most cases, the vast majority of vertices have a very low degree while some have a large degree. This is often captured by the fact that the degree distribution, *i.e.* the proportion p_k for each k of vertices of degree k, is well fitted by a power-law: $p_k \sim k^{-\alpha}$ for an exponent α generally between 2 and 3. See [28, 9, 1, 22, 29, 14] for extensive lists of cases in which this property was observed [2].

We will see that several algorithms proposed in previous section have provable better performances on such graphs than on general (sparse) graphs.

Let us first note that there are several ways to model real-world power-law distributions; see for instance [13, 11]. We use here one of the most simple and classical ones, namely *continuous power-laws*; choosing one of the others would lead to similar results. In such a distribution, p_k is taken to be equal to $\int_k^{k+1} Cx^{-\alpha}\mathrm{d}x$, where C is a normalization constant. This ensures that p_k is proportional to $k^{-\alpha}$ in the limit where k is large. We must moreover ensure that the sum of the p_k is equal to 1: $\sum_{k=1}^{\infty} p_k = \int_1^{\infty} C\ x^{-\alpha}\mathrm{d}x = C\ \frac{1}{\alpha-1} = 1$. We obtain $C = \alpha - 1$, and finally $p_k = \frac{1}{\alpha-1} \cdot \int_k^{k+1} x^{-\alpha}\mathrm{d}x = k^{-\alpha+1} - (k+1)^{-\alpha+1}$.

Finally, when we talk about power-law graphs in the following, we refer to graphs in which the proportion of vertices of degree k is $p_k = k^{-\alpha+1} - (k+1)^{-\alpha+1}$.

Theorem 6 *Given an adjacency array representation of a power-law graph G with exponent α, Algorithm 2 (*compact-forward*) and Algorithm 3 (*new-listing*) with $K \in \Theta(n^{\frac{1}{\alpha}})$ list all its*

[2] Note that if α is a constant then m is in $\Theta(n)$. It may however depend on n, and should be denoted by $\alpha(n)$. In order to keep the notations simple, we do not use this notation, but one must keep this in mind.

triangles in $O(m \cdot n^{\frac{1}{\alpha}})$ time.

Proof: Let us denote by n_K the number of vertices of degree larger than or equal to K. In a power-law graph with exponent α, this number is given by: $\frac{n_K}{n} = \sum_{k=K}^{\infty} p_k$. We have $\sum_{k=K}^{\infty} p_k = 1 - \sum_{k=1}^{K-1} p_k = 1 - (1 - K^{-\alpha+1}) = K^{-\alpha+1}$. Therefore $n_K = n \cdot K^{-\alpha+1}$.

Let us first prove the result concerning Algorithm 3 (*new-listing*). As shown in the proof of Theorem 5, its time complexity is in $O(n_K \cdot m + m \cdot K)$. The value of K that minimizes this is in $\Theta(n^{\frac{1}{\alpha}})$, and the result for this algorithm follows.

Let us now consider the case of Algorithm 2 (*compact-forward*). The time complexity is the same as the one for Algorithm 1 (*forward*), and we use here the same notations as in the proof of Theorem 1. Recall that the vertices are numbered by decreasing order of their degrees.

Let us study the complexity of the intersection computation (line 3aa in Algorithm 1 (*forward*)). It is in $\Theta(|A[u]| + |A[v]|)$. Recall that, at this point of the algorithm, $A[v]$ is nothing but the set of neighbors of v with number lower than the one of v (and thus their degree is at least equal to $d(v)$). Therefore, $|A[v]|$ is bounded both by $d(v)$ and the number of vertices of degree at least $d(v)$, *i.e.* $n_{d(v)}$. Likewise, $|A[u]|$ is bounded by $d(u)$ and by $n_{d(v)}$, since $A[u]$ is the set of neighbors of u with degree at least equal to $d(v)$. Moreover, we have $\eta(u) > \eta(v)$ (line 3a of Algorithm 1 (*forward*)), and so $|A[u]| \leq d(u) \leq d(v)$. Finally, both $|A[u]|$ and $|A[v]|$ are bounded by both $d(v)$ and $n_{d(v)}$, and the intersection computation is in $O(d(v) + n_{d(v)})$.

The value K of $d(v)$ such that these two bounds are equal is $K = n^{\frac{1}{\alpha}}$. Then, the computation of the intersection is in $O(K + n_K) = O(n^{\frac{1}{\alpha}})$, and since the number of such computations is bounded by the number of edges (lines 3 and 3a of Algorithm 1 (*forward*)), we obtain the announced complexity. □

Let us insist on the fact that this results is not an average case complexity: it is indeed a worst case complexity guaranteed as long as one considers power-law graphs.

This result improves significantly the known bounds, as soon as α is large enough, and even if $m \in \Theta(n)$. This holds in particular for typical cases met in practice, where α often is between 2 and 3 [9, 1]. It may be seen as an explanation of the fact that Algorithm 1 (*forward*) has very good performances on graphs with heterogeneous degree distributions, as shown experimentally in [25, 26].

We note however that we have no lower bound for the complexity of triangle listing with the assumption that the graph is a power-law one (which we had for general and sparse graphs); actually, we do not even have a proof of the fact that the given bound is tight for the presented algorithms.

7 A typical practical case.

In [25, 26], the authors present a wide set of experiments on both real-world complex networks and some generated using various models, to evaluate the performances of known algorithms in practice. The overall conclusion is that Algorithm 1 (*forward*) performs best on real-world (sparse and power-law) graphs.

We also performed a wide set of experiments [3] which confirmed that Algorithm 1 (*forward*) is very fast and outperforms classical approaches significantly. They however show that, even in the cases where available memory is sufficient for this algorithm, it is outperformed by

[3] Efficient implementations of the algorithms discussed here are provided at [21].

Algorithm 2 (*compact-forward*), which has the advantage of avoiding management of additional data structures.

Note that Algorithm 3 (*new-listing*) suffers from a serious drawback: it relies on the choice of a relevant value for K. Though in theory this is not a problem, in practice it may be quite difficult to determine the best value for K. It depends both on the machine running the program and on the graph under concern. However, as we will see below on a particular representative instance, the precise value of K has little practical impact. In our experiments, with an appropriate value of K the performances of Algorithm 3 (*new-listing*) are similar to, but lower than, the ones of Algorithm 2 (*compact-forward*).

The key point is that Algorithm 2 (*compact-forward*) and Algorithm 3 (*new-listing*), make it possible to manage graphs that were previously out of reach because of space requirements. To illustrate this, we present now a practical case which previous algorithms were unable to handle on our 8 GigaBytes memory machine. This experiment also has the advantage of being representative of what we observed on a wide variety of instances.

The graph we consider here is a *web* graph provided by the *WebGraph* project [7]. It contains all the web pages in the `.uk` domain discovered during a crawl conducted from the 11-th of july, 2005, at 00:51, to the 30-th at 10:56 using *UbiCrawler* [8]. It has $n = 39,459,925$ vertices and $m = 783,027,125$ (undirected) edges, leading to more than 6 GigaBytes of memory usage if stored in (sorted) (uncompressed) adjacency arrays, each vertex being encoded in 4 bytes as an integer between 0 and $n-1$. Its degree distribution is plotted in Figure 1 (left), showing that the degrees are very heterogeneous and reasonably well fitted by a power-law of exponent $\alpha = 2.5$. It contains $304,529,576$ triangles.

Algorithm 1 (*forward*), as well as the ones based on adjacency matrices, are unable to manage this graph on our 8 GigaBytes memory machine. Instead, and despite the fact that it is quite slow, *edge-iterator*, with its $\Theta'(2m+n)$ space complexity, can handle it. It took approximately 41 hours to count the triangles in this graph with this algorithm on our machine.

Algorithm 2 (*compact-forward*) achieves much better results: it took approximately 20 minutes. Likewise, Algorithm 3 (*new-listing*) took around 45 minutes (depending on the value of K, see Figure 1 (right)). This is probably close to what Algorithm 1 (*forward*) would achieve in 16 GigaBytes of central memory. All the experiments we conducted were consistent with these observations.

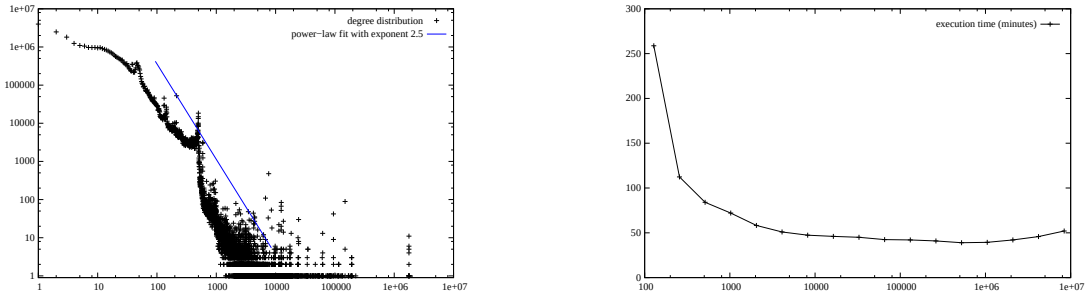

Figure 1: Left: the degree distribution of our graph. Right: the execution time (in minutes) as a function of the number of vertices with degree larger than K.

We plot in Figure 1 (right) the running time of Algorithm 3 (*new-listing*) as a function of the number of vertices with degree larger than K, for varying values of K. Surprisingly enough, this plot shows clearly that the time performance increases drastically as soon as a

few vertices are considered as high degree ones, and then stays stable for wide ranges of K. This may be seen as a consequence of the fact that *edge-iterator* is very efficient when the maximal degree is bounded. In other words, the few high degree vertices are responsible for the low performances of *edge-iterator*.

8 Conclusion.

In this contribution, we push significantly further the limits attainable in practical triangle computations on huge graphs. To achieve this, we first note that space requirements are a key issue in this context, which received surprisingly little attention until now. We study in depth a very efficient algorithm proposed recently and give a more compact version of it. We propose another approach that reaches similar space needs, with the same time complexity.

These two algorithms outperform significantly all previously available ones, either by their time complexity, their space requirements, or both. We prove moreover that they perform better on power-law graphs, which are often met in practice. This makes them particularily suitable for complex network analysis, in which triangle computations on huge power-law graphs play an important role. We illustrate this by computing triangles in a typical real-world complex network, which shows that our goal is reached and that reducing space requirements does indeed have a practical impact.

Moreover, several theoretical and practical problems arised during this study. For instance, the existence of an algorithm able to list triangles in $\Theta(m^{\frac{3}{2}})$ time and $\Theta'(2m+n)$ space (*i.e.* with no significant space in addition to the graph representation) is open. Likewise, designing compact algorithms for *finding* and *counting* faster than using *listing* algorithms is a relevant direction for further research. Obtaining lower bounds and better algorithms for the power-law case would also be very interesting.

Let us notice that other approaches exist, based for instance on streaming algorithmics (avoiding to store the graph in central memory) [17, 4, 19] and/or approximate algorithms [24, 19, 27], and/or various methods to compress the graph [5, 6]. These approaches are very promising for graphs even larger than the ones considered here, in particular the ones that do not fit in central memory.

Another interesting approach would be to express the complexity of triangle algorithms in terms of the number of triangles in the graph (and of its size). Indeed, it may be possible to achieve much better performance for *listing* if the graph contains few triangles. Likewise, it is reasonable to expect that *listing*, but also *counting*, may perform poorly if there are many triangles in the graph. The *finding* problem, on the countrary, may be easier on graphs having many triangles. To our knowledge, this direction has not yet been explored.

Finally, the results we present in Section 6 take advantage of the fact that most very large graphs considered in practice may be approximed by power-law graphs. It is not the first time that algorithms for triangle problems use underlying graph properties to get improved performance. It however appeared quite recently that many large graphs met in practice have some nontrivial (statistical) properties in common, and using these properties in the design of efficient algorithms still is at its very beginning. We consider this as a key direction for further research.

Acknowledgments. I warmly thank Frédéric Aidouni, Michel Habib, Vincent Limouzy, Clémence Magnien, Thomas Schank and Pascal Pons for helpful comments and references. This work was partly funded by the MetroSec (Metrology of the Internet for Security) and AGRI (Analyse des Grands Réseaux d'Interaction) projects.

References

[1] R. Albert and A.-L. Barabási. Statistical mechanics of complex networks. *Reviews of Modern Physics*, 74, 47, 2002.

[2] Noga Alon, Raphael Yuster, and Uri Zwick. Finding and counting given length cycles. In *European Symposium Algorithms (ESA)*, 1994.

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

[4] Ziv Bar-Yossef, Ravi Kumar, and D. Sivakumar. Reduction in streaming algorithms with an application of counting triangles in graphs. In *ACM/SIAM Symposium On Discrete Algorithms (SODA)*, 2002.

[5] P. Boldi and S. Vigna. The webgraph framework i: compression techniques. In *WWW*, 2004.

[6] P. Boldi and S. Vigna. The webgraph framework ii: Codes for the world-wide web. In *DCC*, 2004.

[7] Paolo Boldi. WebGraph project. `http://webgraph.dsi.unimi.it/`.

[8] Paolo Boldi, Bruno Codenotti, Massimo Santini, and Sebastiano Vigna. Ubicrawler: a scalable fully distributed web crawler. *Softw., Pract. Exper.*, 34(8):711–726, 2004.

[9] U. Brandes and T. Erlebach, editors. *Network Analysis: Methodological Foundations*. LNCS, Springer-Verlag, 2005.

[10] Norishige Chiba and Takao Nishizeki. Arboricity and subgraph listing algorithms. *SIAM Journal of Computing*, 14, 1985.

[11] R. Cohen, R. Erez, D. ben Avraham, and S. Havlin. Reply to the comment on 'breakdown of the internet under intentional attack'. *Phys. Rev. Lett*, 87, 2001.

[12] Don Coppersmith and Shmuel Winograd. Matrix multiplication via arithmetic progressions. *J. Symb. Comput.*, 9(3):251–280, 1990.

[13] S.N. Dorogovtsev and J.F.F. Mendes. Comment on 'breakdown of the internet under intentional attack'. *phys. Rev. Lett*, 87, 2001.

[14] Stephen Eubank, V.S. Anil Kumar, Madhav V. Marathe, Aravind Srinivasan, and Nan Wang. Structural and algorithmic aspects of massive social networks. In *ACM/SIAM Symposium on Discrete Algorithms (SODA)*, 2004.

[15] Frank Harary and Helene J. Kommel. Matrix measures for transitivity and balance. *Journal of Mathematical Sociology*, 1979.

[16] 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.

[17] Monika Rauch Henzinger, Prabhakar Raghavan, and Sridar Rajagopalan. Computing on data streams. Technical report, DEC Systems Research Center, 1998.

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

[19] H. Jowhari and M. Ghodsi. New streaming algorithms for counting triangles in graphs. In *CO-COON*, 2005.

[20] Matthieu Latapy. Theory and practice of triangle problems in very large (sparse (power-law)) graphs. Technical Report. `http://www.liafa.jussieu.fr/~latapy/Publis/triangles.pdf`.

[21] Matthieu Latapy. Triangle computation web page. `http://www.liafa.jussieu.fr/~latapy/Triangles/`.

[22] 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, 2002.

[23] Burkhard Monien. How to find long paths efficiently. *Annals of Discrete Mathematics*, 25:239–254, 1985.

[24] Thomas Schank and Dorothea Wagner. Approximating clustering coefficient and transitivity. *Journal of Graph Algorithms and Applications (JGAA)*, 9:2:265–275, 2005.

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

[26] Thomas Schank and Dorothea Wagner. Finding, counting and listing all triangles in large graphs, an experimental study. In *Workshop on Experimental and Efficient Algorithms (WEA)*, 2005.

[27] Asaf Shapira and Noga Alon. Homomorphisms in graph property testing - a survey. In *Electronic Colloquium on Computational Complexity (ECCC)*, 2005.

[28] Duncan J. Watts and Steven H. Strogatz. Collective dynamics of smallworld networks. *Nature*, 393:440–442, 1998.

[29] Esti Yeger-Lotem, Shmuel Sattath, Nadav Kashtan, Shalev Itzkovitz, Ron Milo, Ron Y. Pinter, and Uri Alon andHanah Margalit. Network motifs in integrated cellular networks of transcription-regulation and protein-protein interaction. *Proc. Natl. Acad. Sci. USA 101*, pages 5934–5939, 2004.