Contents lists available at ScienceDirect

Theoretical Computer Science

journal homepage: www.elsevier.com/locate/tcs

# Main-memory triangle computations for very large (sparse (power-law)) graphs

Matthieu Latapy

*LIP6, CNRS and Université Pierre et Marie Curie, 4 place Jussieu, 75005 Paris, France*

ARTICLE INFO

*Article history:*
Received 7 December 2006
Received in revised form 1 February 2008
Accepted 23 July 2008
Communicated by G. Italiano

*Keywords:*
Graphs
Algorithms
Triangles
Complex networks

ABSTRACT

Finding, counting and/or listing triangles (three vertices with three edges) in massive graphs are natural fundamental problems, which have recently received much attention because of their importance in complex network analysis. Here we provide a detailed survey of proposed main-memory solutions to these problems, in a unified way.

We note that previous authors have paid surprisingly little attention to space complexity of main-memory solutions, despite its both fundamental and practical interest. We therefore detail space complexities of known algorithms and discuss their implications. We also present new algorithms which are time optimal for triangle listing and beats previous algorithms concerning space needs. They have the additional advantage of performing better on power-law graphs, which we also detail. We finally show with an experimental study that these two algorithms perform very well in practice, allowing us to handle cases which were previously out of reach.

## 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. Following classical conventions, we call *finding*, *counting* and *listing* the problems of deciding if a given graph contains any triangle, counting the number of triangles in the graph, and listing all of them, respectively. We moreover call *node-counting* the problem of counting *for each vertex* the number of triangles to which it belongs. We refer to all these problems as a whole by *triangle problems*.

Triangle problems may be considered as classical, natural and fundamental algorithmic questions, and have been studied as such [24,14,2,3,33,34].

Moreover, they have recently gained much practical importance since they are central in the so-called *complex network analysis*, see for instance [36,13,1,19]. First, they are involved in the computation of one of the main statistical property used to describe large graphs met in practice, namely the clustering coefficient [36]. The clustering coefficient of a vertex $v$ (of degree at least 2) is the probability that any two randomly chosen neighbors of $v$ are linked together. It is computed by dividing the number of triangles containing $v$ by the number of possible edges between its neighbors, i.e. $\binom{d(v)}{2}$ if $d(v)$ denotes the number of neighbors of $v$. One may then define the clustering coefficient of the whole graph as the average of this value for all the vertices (of degree at least 2). Likewise, the *transitivity ratio* [1] [22,21] is defined as $\frac{3N_\Delta}{N_\vee}$ where $N_\Delta$ denotes the number of triangles in the graph and $N_\vee$ denotes the number of connected triples, i.e. pairs of edges with one common extremity, in the graph.

*E-mail address:* Matthieu.Latapy@lip6.fr.

[1] Even though some authors make no distinction between the two notions, they are different, see for instance [12,32]. Both have their own advantages and drawbacks, but discussing this is out of the scope of this contribution.

0304-3975/$ – see front matter © 2008 Elsevier B.V. All rights reserved.
doi:10.1016/j.tcs.2008.07.017

In the context of complex network analysis, triangles also play a key role in the study of motif occurrences, i.e. the presence of special (small) subgraphs in given (large) graphs. This has been studied in particular in protein interaction networks, where some motifs may correspond to biological functions, see for instance [29,37]. Triangles often are key parts of such motifs.

In summary, triangle finding, counting, node-counting and/or listing appear as key issues both from a fundamental point of view and for practical purpose. The aim of this contribution is to review the algorithms proposed until now for solving these problems with both a fundamental perspective (we discuss asymptotic complexities and give detailed proofs) and a practical one (we discuss space requirements and graph encoding, and we evaluate algorithms with some experiments).

We note that, until now, authors have paid surprisingly little attention to space requirements of main-memory algorithms for triangle problems; this however is an important limitation in practice, and this also induces interesting theoretical questions. We therefore discuss this (all space complexity results stated in this paper are new, though very simple in most cases), and we present space-efficient algorithms.

Approaches relying on streaming algorithms [23,4,25], approximate results [32,25,35], or compressed graphs [8,9] also exist. They are of high interest in cases where the graphs do not fit in main-memory. Otherwise, they are much slower and more intricate than main-memory algorithms, on which we focus here. We will see that, as long as the graph fits in main-memory, such approaches are sufficient.

The paper is organised as follows. After a few preliminaries (Section 2), we begin with results on finding, counting and node-counting problems, among which basically no difference in complexity is known (Section 3). Then we turn to the harder problem of triangle listing, in Section 4. In these parts of the paper, we deal with both the general case (no assumption is made on the graph) and on the important case where the graph is sparse. Many very large graphs met in practice also have heterogeneous degrees; we focus on this case in Section 5. Finally, we present experimental evaluations in Section 6. We summarise the current state-of-the-art and we point out the main perspectives in Section 7.

## 2. Preliminaries

Throughout the paper, we consider an undirected[2] 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 also assume that $m \in \Omega(n)$; this is a classical convention which plays no role in our algorithms but makes complexity formulae simpler. 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 $G$: $d_{\max} = \max_v\{d(v)\}$.

Before entering in the core of this paper, we need to discuss a few issues that play an important role in the following. They are necessary to make the discussion all along the paper precise and rigorous.

*Notation for precise space complexity*

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. We will give an illustration of this in Section 6.

In order to capture this situation, 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 space cost of the algorithm is exactly $\sigma f(n, m) + c$ where $c$ is any constant and $\sigma$ 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 $\sigma$ 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.

Throughout this paper, we will assume that graphs are given by their adjacency matrix and/or adjacency array representation. Results on such representations may easily be converted into results on adjacency list representations (only the space complexity in terms of $\Theta'()$ is affected), as well as more subtle adjacency representations (hashtables or balanced trees for instance).

Each adjacency array in an adjacency array representation of $G$ may moreover be sorted. This can be done in place in $\Theta(m \log(n))$ time and $\Theta'(2m + n) \subseteq \Theta(m)$ space (only a constant space is needed in addition to the representation of $G$).[3]

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

[2] i.e. we make no distinction between $(u, v)$ and $(v, u)$ in $V \times V$.

[3] Even better performance may be obtained using (compact) radix sorting, see for instance [20]. The improvement however plays no significant role in our context, therefore we do not discuss this further.

enter in these details only in cases where the time and space complexities are small enough to make the precise space cost interesting.

*Worst case complexity, and graph families*

All the complexities we discuss in this paper are *worst case* complexities, in the sense that they are bounds for the time and space needs of the algorithms, on any input. In most cases, these bounds are tight (leading to the use of the $\Theta()$ notation, see for instance [17] for definitions). In other words, we say that an algorithm is in $\Theta(f(n))$ if for all possible instances of the input the algorithm runs within this complexity, and there is at lease one instance for which it is reached. In several cases, however, the worst case complexity is actually the complexity for any input (in the case of Theorem 4, for instance, and for most space complexities).

It would also be of high interest to study the *expected* behavior of triangle algorithms, in addition to the worst case one. This has been done in some cases; for instance, it is proved in [24] that *vertex-iterator* (see Section 4.1) has expected time complexity in $O(n^{\frac{5}{3}})$. Obtaining such results however is often very difficult, and their relevance for practical purposes is not always clear: the choice of a model for the average input is a difficult task (in our context, random graphs would be an unsatisfactory choice [13,1,36]). We therefore focus on worst case analysis, which has the advantage of giving guarantees on the behaviors of algorithms, on any input.

Another interesting approach is to study (worst case) complexities on given graph families. This has already been done on various cases, the most important ones probably being the sparse graphs, i.e. graphs in which $m$ is in $o(n^2)$. This is motivated by the fact that most real-world complex networks lead to such graphs, see for instance [13,1,36]. Often, it is even assumed that $m$ is in $O(n)$. Recent studies however show that, despite the fact that $m$ is small compared to $n^2$, it may be in $\omega(n)$ [28,31,27]. Other classes of graphs have been considered, like for instance planar graphs: it is shown in [24] that one may decide if any planar graph contains a triangle in $O(n)$ time.

We do not detail all these results here. Since we are particularly interested in real-world complex networks, we present in detail the results concerning sparse graphs all along the paper. We also introduce new results on power-law graphs (Section 5), which capture an important property met in practice. A survey on available results on specific classes of graphs remains to be done, and is out of the scope of this paper.

## 3. The fastest algorithms for finding, counting, and node-counting

The fastest algorithm known for node-counting relies on fast matrix product [24,2,3,16]. Indeed, if one considers the adjacency matrix $A$ of $G$ then the value $A^3_{vv}$ on the diagonal of $A^3$ is nothing but twice the number of triangles to which $v$ belongs, for any $v$. Finding, counting and node-counting triangle problems can therefore be solved in $O(n^\omega)$ time, where $\omega < 2.376$ is the fast matrix product exponent [16]. This was first noticed in 1978 [24], and currently no faster algorithm is known for any of these problems in the general case, even for triangle finding (but this is no longer true when the graph is sparse, see Theorem 2).

This approach naturally needs the graph to be given by its adjacency matrix representation. Moreover, it makes it necessary to compute and store the matrix $A^2$, leading to a $\Theta(n^2)$ space complexity in addition to the adjacency matrix storage.

**Theorem 1** (*[24,16]*). *Given the adjacency matrix representation of G, it is possible to solve triangle finding, counting and node-counting in $O(n^\omega) \subset O(n^{2.376})$ time and $\Theta(n^2)$ space on G using fast matrix product.*

This time complexity is the current state of our knowledge, as long as one makes no assumption on $G$. Note that no non-trivial lower bound is known for this complexity; therefore faster algorithms may be designed.

As we will see, there exist (slower) algorithms with lower space complexity for these problems. Some of these algorithms only need an adjacency array representation of $G$. They are derived from listing algorithms, which we present in Section 4.

One can design faster algorithms if $G$ is sparse. In [24], it was first proved that triangle finding, counting, node-counting and listing[4] can be solved in $\Theta(m^{\frac{3}{2}})$ time and $\Theta(m)$ space. This result has been improved in [14] using a property of the graph (namely arboricity) but the worst case complexities were unchanged. No better result was known until 1995 [3,2], where the authors prove Theorem 2,[5] which constitutes a significant improvement although it relies on very simple ideas. We detail the proof and give a slightly different version, which will be useful in the following (similar ideas are used in Section 4.3, and this proof permits a straightforward extension of this theorem in Section 5).

[4] The original results actually concern triangle *finding* but they can easily be extended to counting, node-counting and listing at no cost; we present such an extension in Section 4, Algorithm 4 (*tree-listing*).

[5] Again, the original results concerned triangle *finding*, but may easily be extended to node-counting, see Algorithm 1 (*ayz-node-counting*), and listing, see Algorithm 5 (*ayz-listing*). This was first proposed in [33,34]. These algorithms have also been generalized to longer cycles in [38] but this is out of the scope of this paper.

**Algorithm 1 – *ayz-node-counting.*** *Counts for all $v$ the triangles in G containing $v$ [3,2].*

```
Input: the adjacency array representation of $G$, its adjacency matrix $A$, and an integer $K$
Output: $T$ such that $T[v]$ is the number of triangles in $G$ containing $v$
1. initialise $T[v]$ to 0 for all $v$
2. for each vertex $v$ with $d(v) \leq K$:
 2a. for each pair $\{u, w\}$ of neighbors of $v$:
  2aa. if $A[u, w]$ then:
   2aaa. increment $T[v]$
   2aab. if $d(u) > K$ and $d(w) > K$ then increment $T[u]$ and $T[w]$
   2aac. else if $d(u) > K$ and $w > v$ then increment $T[u]$
   2aad. else if $d(w) > K$ and $u > v$ then increment $T[w]$
3. let $G'$ be the subgraph of $G$ induced by $\{v,\ d(v) > K\}$
4. construct the adjacency matrix $A'$ of $G'$
5. compute $A'^3$ using fast matrix product
6. for each vertex $v$ with $d(v) > K$:
 6a. add to $T[v]$ half the value in $A'^3_{vv}$
```

**Theorem 2** ([3,2]). *Given the adjacency array representation of G and its adjacency matrix, it is possible to solve triangle finding, counting and node-counting on G in $O(m^{\frac{2\omega}{\omega+1}}) \subset O(m^{1.41})$ time and $\Theta\left(n^2\right)$ space; Algorithm 1 (*ayz-node-counting*) achieves this if one takes $K \in \Theta(m^{\frac{\omega-1}{\omega+1}})$.*

**Proof.** Let us first show that Algorithm 1 (*ayz-node-counting*) solves node-counting (and thus counting and finding). Consider a triangle in $G$ that contains a vertex with degree at most $K$; then it is discovered in lines 2a and 2aa. Lines 2aaa to 2aad ensure that it is counted exactly once for each vertex it contains. Consider now the triangles in which all the vertices have degree larger than $K$. Each of them induces a triangle in $G'$, and $G'$ contains no other triangle. These triangles are counted using the matrix product approach (lines 5, 6 and 6a), and finally all the triangles in $G$ are counted for each vertex.

Let us now study the time complexity of Algorithm 1 (*ayz-node-counting*) in function of $K$. For each vertex $v$ with $d(v) \leq K$, one counts the number of triangles containing $v$ in $\Theta(d(v)^2) \subseteq O(d(v)K)$ thanks to the adjacency array representation of $G$. If we sum over all the vertices in the graph this leads to a time complexity in $O(mK)$ for lines 2 to 2aad. Now notice that there cannot be more than $\frac{2m}{K}$ vertices $v$ with $d(v) > K$. Line 4 constructs (in $O\left(m + (\frac{m}{K})^2\right)$ time, which plays no role in the global complexity) the adjacency matrix of the subgraph $G'$ of $G$ induced by these vertices. Using fast matrix product, line 5 computes the number of triangles for each vertex in $G'$ in time $O\left(\left(\frac{m}{K}\right)^{\omega}\right)$. Finally, we obtain the overall time complexity of the algorithm: $O\left(mK + \left(\frac{m}{K}\right)^{\omega}\right)$.

In order to minimize this, one has to search for a value of $K$ such that $mK \in \Theta((\frac{m}{K})^{\omega})$. This leads to $K \in \Theta(m^{\frac{\omega-1}{\omega+1}})$, which gives the announced time complexity.

The space complexity comes directly from the need of the adjacency matrix. All other space costs are lower; in particular, the graph $G'$ may contain $\frac{2m}{K}$ vertices, which leads to a $\Theta\left(\left(\frac{m}{K}\right)^2\right) = \Theta\left(m^{2\left(1-\frac{\omega-1}{\omega+1}\right)}\right) = \Theta\left(m^{\frac{4}{\omega+1}}\right) \subset O(m^{1.185})$ space cost for $A'$, $A'^2$ and $A'^3$. □

Note that one may also use *sparse* matrix product algorithms, see for instance [39]. However, the matrix $A^2$ may not be sparse (in particular if there are vertices with large degrees, which is often the case in practice as discussed in Section 5). But algorithms may take benefit from the fact that *one* of the two matrices involved in a product is sparse, and there also exists algorithms for products of more than two sparse matrices. These approaches lead to situations where it depends on the precise relation between $n$ and $m$ which algorithm is the fastest. Discussing this further therefore is quite complex, and it is out of the scope of this paper.

In conclusion, despite the fact that the algorithms presented in this section are asymptotically very fast, they have two important limitations. First, they have a prohibitive space cost, since the matrices involved in the computation (in addition to the adjacency matrix, but it is considered as the encoding of $G$ itself) may need $\Theta(n^2)$ space. Moreover, the fast matrix product algorithms are quite intricate, which leads to difficult implementations with high risks of errors. This also leads to large constant factors in the complexities, which have no importance at the asymptotic limit but may play a significant role in practice.

For these reasons, and despite the fact that they clearly are of prime theoretical importance, these algorithms have limited practical impact. Instead, one generally uses one of the *listing* algorithms (adapted accordingly) which we detail now.

## 4. Time-optimal listing algorithms

First notice that there may be $\binom{n}{3} \in \Theta(n^3)$ triangles in $G$. Likewise, there may be $\Theta(m^{\frac{3}{2}})$ triangles, since $G$ may be a clique of $\sqrt{m}$ vertices (thus containing $\binom{\sqrt{m}}{3} \in \Theta(m^{\frac{3}{2}})$ triangles). This gives the following lower bounds for the time complexity of any triangle listing algorithm.

**Lemma 3** (*[24,33,34]*). *Listing all triangles in $G$ is in $\Omega(n^3)$ and $\Omega(m^{\frac{3}{2}})$ time.*

In this section, we first observe that the time complexity $\Theta(n^3)$ can easily be reached (Section 4.1). However, $\Theta(m^{\frac{3}{2}})$ is much better in the case of sparse graphs. We present more subtle algorithms that reach this bound (Section 4.2). Again, space complexity is a key issue, and we discuss this for each algorithm. We will see that algorithms proposed until now rely on the use of adjacency matrices and/or need $\Omega(m)$ space in addition to the graph encoding. We improve this by proposing algorithms with lower space needs, using only the adjacency array representation of $G$, and still in $\Theta(m^{\frac{3}{2}})$ time (Section 4.3).

### *4.1. Basic algorithms*

One may trivially obtain a listing algorithm in $\Theta(n^3)$ (optimal) time with the matrix representation of $G$ by testing in $\Theta(1)$ time any possible triple of vertices. Moreover, this algorithm needs only a constant (and very small) amount of space in addition to the graph representation.

**Theorem 4** (*[33,34] and Folklore*). *Given the adjacency matrix representation of $G$, it is possible to solve triangle listing in $\Theta(n^3)$ time and $\Theta(n^2)$ space using the direct testing of every triple of vertices.*

This approach however has severe drawbacks. First, it needs the adjacency matrix of $G$. More importantly, its complexity does not depend on the actual properties of $G$; it always needs $\Theta(n^3)$ computation steps even if the graph contains very few edges. It must however be clear that, if almost all triples of vertices form a triangle, no better asymptotic bound can be attained, and the simplicity of this algorithm makes it very efficient in these cases.

In order to obtain faster algorithms on sparse graphs, while keeping the implementation very simple, one often uses the following algorithms. The first one, introduced in [24] and called *vertex-iterator* in [33,34], consists in iterating Algorithm 2 (*vertex-listing*) on each vertex of $G$. The second one, which seems to be the most widely used algorithm[6], consists in iterating Algorithm 3 (*edge-listing*) over each edge in $G$. It was first introduced in [24], and discussed in [33,34] where the authors call it *edge-iterator*.

**Algorithm 2 – *vertex-listing.*** *Lists all the triangles containing a given vertex [24].*

```
Input: the adjacency array representation of $G$, its adjacency matrix $A$, and a vertex $v$
Output: all the triangles to which $v$ belongs
1. for each pair $\{u, w\}$ of neighbors of $v$:
  1a. if $A_{uw} = 1$ then output triangle $\{u, v, w\}$
```

**Algorithm 3 – *edge-listing.*** *Lists all the triangles containing a given edge [24].*

```
Input: the sorted adjacency array representation of $G$, and an edge $(u, v)$ of $G$
Output: all the triangles in $G$ containing $(u, v)$
1. for each $w$ in $N(u) \cap N(v)$:
  1a. output triangle $\{u, v, w\}$
```

**Theorem 5** (*[24,33,34]*). *Given the adjacency array representation of $G$ and its adjacency matrix, it is possible to list all its triangles in $\Theta\left(\sum_v d(v)^2\right)$, $\Theta(md_{\max})$, $\Theta(mn)$, and $\Theta(n^3)$ time and $\Theta(n^2)$ space;* vertex-iterator *achieves this.*

**Proof.** The fact that Algorithm 2 (*vertex-listing*) lists all the triangles to which a vertex $v$ belongs is straightforward. Then, iterating over all vertices gives three times each triangle; if one wants each triangle only once it is sufficient to restrict the output of triangles to the ones for which $\eta(w) > \eta(v) > \eta(u)$, for any injective numbering $\eta()$ of the vertices.

Thanks to the adjacency array representation of $G$, the pairs of neighbors of $v$ may be computed in $\Theta(d(v)^2)$ time and $\Theta(1)$ space (this would be impossible with the adjacency matrix only). Thanks to the adjacency matrix, the test in line 1a may be processed in $\Theta(1)$ time and space (this would be impossible with the adjacency array representation only). The time complexity of Algorithm 2 (*vertex-listing*) therefore is in $\Theta(d(v)^2)$ time and $\Theta(1)$ space. The $\Theta(\sum_v d(v)^2)$ time and $\Theta(n^2)$ space complexity of the overall algorithm follows. Moreover, we have $\Theta(\sum_v d(v)^2) \subseteq O(\sum_v d(v)d_{\max}) = O(md_{\max}) \subseteq O(mn) \subseteq O(n^3)$, and all these complexity may be attained in the worst case (clique of $n$ vertices), hence the results. □

[6] It is for instance implemented in the widely used complex network analysis software *Pajek* [7,6,5].

**Theorem 6** (*[24,33,34] and Folklore*). *Given the sorted adjacency array representation of G, it is possible to list all its triangles in $\Theta(md_{\max})$, $\Theta(mn)$ and $\Theta(n^3)$ time and $\Theta'(2m+n) \subseteq \Theta(m)$ space; The* edge-iterator *algorithm achieves this.*

**Proof.** The correctness of the algorithm is immediate. One may proceed like in the proof of Theorem 5 to obtain each triangle only once.

Each edge $(u, v)$ is treated in time $\Theta(d(u) + d(v))$ (because $N(u)$ and $N(v)$ are sorted) and $\Theta(1)$ space. We have $d(u) + d(v) \in \Theta(d_{\max})$, therefore the overall time complexity is in $O(md_{\max}) \subseteq O(mn) \subseteq O(n^3)$. In the worst case (clique of $n$ vertices) all these complexity are tight.

The space complexity is nothing but the one of the graph representation, as the algorithms needs only a constant additional space. □

First note[7] that these algorithms are optimal in the worst case, just like the direct method (Lemma 3 and Theorem 4). However, they are much more efficient on sparse graphs, in particular if the maximal degree is low [7], since they both are in $\Theta(md_{\max})$ time. If the maximal degree is a constant, *vertex-iterator* even is in $\Theta(n)$ time (like *edge-iterator*). Moreover, both algorithms only need a constant amount of space in addition to the graph encoding, which makes them very interesting from this perspective.

However, *vertex-iterator* has a severe drawback: it needs the adjacency matrix of $G$ *and* the adjacency array representation. Instead, *edge-iterator* only needs the sorted adjacency array representation, which is often available in practice.[8] Moreover, *edge-iterator* runs in $\Theta'(2m+n) \subseteq \Theta(m)$ space, which makes it very compact. Because of these two reasons, and because of its simplicity, it is widely used in practice.

The performance of these algorithms however are quite poor when the maximal degree is unbounded, and in particular if it grows like a power of $n$. They may even be asymptotically sub-optimal on sparse graphs and/or on graphs with some vertices of high degree, which often appear in practice (we discuss this further in Section 5). It is however possible to design time-optimal listing algorithms for sparse graphs, which we detail now.

### 4.2. Time-optimal listing algorithms for sparse graphs

Several algorithms have been proposed that reach the $\Theta(m^{\frac{3}{2}})$ bound of Lemma 3, and thus are time optimal on sparse graphs (note that this is also optimal for dense graphs, but we have seen in Section 4.1 much simpler algorithms for these cases). In 1978, an algorithm was proposed to *find* a triangle in $\Theta(m^{\frac{3}{2}})$ time and $\Theta(n^2)$ space [24]. Therefore, it is slower than the ones discussed in Section 3 for finding, but it may be extended to obtain a *listing* algorithm with the same complexity. We first present this below. Then, we detail two simpler solutions with this complexity, proposed recently in [33,34]. The first one consists in a simple extension of Algorithm 1 (*ayz-node-counting*); the other one, named *forward*, has the advantage of being very efficient in practice [33,34]. Moreoever, it has a much lower space complexity. In addition, we will slightly modify it in Section 4.3 to reach a $\Theta'(2m+2n) \subseteq \Theta(m)$ space cost, which makes it very compact.

*An approach based on covering trees [24]*

We use here the classical notions of covering trees and connected components, as defined for instance in [17]. Since they are very classical, we do not recall them. We just note that a covering tree of each connected component of any graph may be computed in time linear in the number of edges of this graph, and space linear in its number of vertices (typically using a breadth-first search). One then has access to the father of any vertex in $\Theta(1)$ time and space.

In [24], the authors propose a triangle *finding* algorithm in $\Theta(m^{\frac{3}{2}})$ time and $\Theta(n^2)$ space. We present here a simple extension of this algorithm to solve triangle *listing* with the same complexity. To achieve this, we need the following lemma, which is a simple extension of Lemma 4 in [24].

**Lemma 7** (*[24]*). *Let us consider a covering tree for each connected component of G, and a triangle t in G having an edge in one of these trees. Then there exists an edge $(u, v)$ in E but in none of these trees, such that $t = \{u, v, father(v)\}$.*

**Proof.** Let $t = \{x, y, z\}$ be a triangle in $G$, and let $T$ be the tree that contains an edge of $t$. We can suppose without loss of generality that this edge is $(x, y = \text{father}(x))$. Two cases have to be considered. First, if $(x, z) \notin T$ then it is in none of the trees, and taking $v = x$ and $u = z$ satisfies the claim. Second, if $(x, z) \in T$ then we have $\text{father}(z) = x$ (because $\text{father}(x) = y \neq z$). Moreover, $(y, z) \notin T$ (else $T$ would contain a cycle, namely $t$). Therefore taking $v = z$ and $u = y$ satisfies the claim. □

This lemma shows that, given a covering tree of each connected component of $G$, one may find triangles by checking for each edge $(u, v)$ that belongs to none of these trees if $\{u, v, \text{father}(v)\}$ is a triangle. Then, all the triangles containing $(v, \text{father}(v))$ are discovered. This leads to Algorithm 4 (*tree-listing*), and to the following result (which is a direct extension of the one concerning triangle *finding* described in [24]).

[7] We also note that another $O(mn)$ time algorithm was proposed in [30] for a more general problem. In the case of triangles, it does not improve *vertex-iterator* and *edge-iterator*, which are much simpler, therefore we do not detail it here.

[8] Recall that, if needed, one may sort the adjacency array representation of $G$ in $\Theta(m \log(n))$ time and using only a constant amount of space in addition to the one needed for the graph representation.

**Algorithm 4 – *tree-listing.*** *Lists all the triangles in a graph [24].*

```
Input: the adjacency array representation of $G$, and its adjacency matrix $A$
Output: all the triangles in $G$
1. while there remains an edge in $E$:
  1a. compute a covering tree for each connected component of $G$
  1b. for each edge $(u, v)$ in none of these trees:
    1ba. if $(\text{father}(u), v) \in E$ then output triangle $\{u, v, \text{father}(u)\}$
    1bb. else if $(\text{father}(v), u) \in E$ then output triangle $\{u, v, \text{father}(v)\}$
  1c. remove from $E$ all the edges in these trees
```

**Theorem 8** ([24]). *Given the adjacency array representation of G and its adjacency matrix, it is possible to list all its triangles in $\Theta(m^{\frac{3}{2}})$ time and $\Theta(n^2)$ space; Algorithm 4 (*tree-listing*) achieves this.*

**Proof.** Let us first prove that the algorithm is correct. It is clear that the algorithm may only output triangles. Suppose that one is missing. But all its edges have been removed when the computation stops, and so (at least) one of its edges was in a tree at some step. Let us consider the first such step (therefore the three edges of the triangle are present). Lemma 7 says that there exists an edge satisfying the condition tested in lines 1b and 1ba, and thus the triangle was discovered at this step. Finally, we reach a contradiction, and thus all triangles have been discovered.

Now let us focus on the time complexity. Following [24], let $c$ denote the number of connected components at the current step of the algorithm. The value of $c$ increases during the computation, until it reaches $c = n$. Two cases have to be considered. First suppose that $c \leq n - \sqrt{m}$. During this step of the algorithm, $n - c \geq n - (n - \sqrt{m}) = \sqrt{m}$ edges are removed. And thus there can be no more than $\frac{m}{\sqrt{m}} = \sqrt{m}$ such steps. Consider now the other case, $c > n - \sqrt{m}$. The maximal degree then is at most $n - c < n - (n - \sqrt{m}) = \sqrt{m}$, and, since the degree of each vertex (of non-null degree) decreases at each step, there can be no more than $\sqrt{m}$ such steps. Finally, the total number of steps is bounded by $2\sqrt{m}$. Moreover, each step costs $O(m)$ time: the test in line 1ba is in $\Theta(1)$ time thanks to the adjacency matrix, and line 1b finds the $O(m)$ edges on which it is ran in $O(m)$ time thanks to the father() relation which is in $\Theta(1)$ time. This leads to the $O(m^{\frac{3}{2}})$ time complexity, and, from Lemma 3, this bound is tight.

Finally, the space complexity comes from the need of the adjacency matrix in input. □

The performances of this algorithm rely on the fact that the graph is given both in its adjacency matrix representation *and* its adjacency array one. This reduces significantly the practical relevance of this approach concerning reduced space complexity. We will see in the next section algorithms that have the same time complexity but need much less space.

*An extension of Algorithm 1 (*ayz-node-counting*) [3,2,33,34]*

The fastest known algorithm for finding, counting, and node-counting triangles, namely Algorithm 1 (*ayz-node-counting*), was proposed in [3,2] and we detailed it in Section 3. As proposed first in [33,34], it is easy to modify it to obtain a *listing* algorithm, namely Algorithm 5 (*ayz-listing*).

**Algorithm 5 – *ayz-listing.*** *Lists all the triangles in a graph [3,2,33,34].*

```
Input: the adjacency array representation of $G$, its adjacency matrix $A$, and an integer $K$
Output: all the triangles in $G$
1. for each vertex $v$ with $d(v) \leq K$:
  1a. output all triangles containing $v$ with Algorithm 2 (vertex-listing), without duplicates
2. let $G'$ be the subgraph of $G$ induced by $\{v,\ d(v) > K\}$
3. compute the sorted adjacency array representation of $G'$
4. list all triangles in $G'$ using Algorithm 3 (edge-listing)
```

**Theorem 9** ([33,34,3,2]). *Given the adjacency array representation of G and its adjacency matrix, it is possible to list all its triangles in $\Theta(m^{\frac{3}{2}})$ time and $\Theta(n^2)$ space; Algorithm 5 (*ayz-listing*) achieves this if one takes $K \in \Theta(\sqrt{m})$.*

**Proof.** First recall that one may sort the adjacency array representation of $G$ in $O(m\log(n))$ time and $\Theta(1)$ space. This has no impact on the overall complexity of Algorithm 5 (*ayz-listing*), thus we suppose in this proof that the representation is sorted.

In a way similar to the proof of Theorem 2, let us first express the complexity of Algorithm 5 (*ayz-listing*) in terms of $K$. Using the $\Theta(d(v)^2)$ complexity of Algorithm 2 (*vertex-listing*) we obtain that lines 1 and 1a have a cost in $O(\sum_{v,d(v)\leq K} d(v)^2) \subseteq O(\sum_{v,d(v)\leq K} d(v)K \subseteq O(mK)$ time and in $\Theta(1)$ space.

Since we may suppose that the adjacency array representation of $G$ is sorted, line 3 can be achieved in $O(m)$ time. The number of vertices in $G'$ is in $\Theta(\frac{m}{K})$ and it may be a clique, thus the space needed for $G'$ is in $\Theta((\frac{m}{K})^2)$.

Finally, the overall time complexity is in $O\left(mK + m\frac{m}{K}\right)$. The optimum is attained with $K$ in $\Theta(\sqrt{m})$, leading to the announced time complexity (which is tight from Lemma 3). The space needed for $G'$ then is $\Theta((\frac{m}{K})^2) = \Theta(m)$, and thus the algorithm has $\Theta(n^2)$ space complexity due to the adjacency array needed in input. □

Again, this result has a significant space cost: it needs the adjacency matrix of $G$, and, even then, it needs $\Theta(m)$ additional space. Moreover, it relies on the use of a parameter, $K$, which may be difficult to choose in practice: though Theorem 9 says that it must be in $\Theta(\sqrt{m})$, this makes little sense when one considers a given graph. We discuss further this issue in Section 6.

*The* forward *fast algorithm [33,34]*

In [33,34], the authors propose another algorithm with optimal time complexity and a $\Theta(m)$ space cost (it needs the adjacency array representation of $G$ only). We now present it in detail. We give a new proof of the correctness and complexity of this algorithm, in order to be able to extend it in the next sections (in particular in Section 5).

**Algorithm 6 – *forward*.** *Lists all the triangles in a graph [33,34].*

```
Input:  the the adjacency array representation of $G$
Output:  all the triangles in $G$
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 $A[u]$
```

**Theorem 10** (*[33,34]*). *Given the adjacency array representation of G, it is possible to list all its triangles in $\Theta(m^{\frac{3}{2}})$ time and $\Theta'(3m + 3n) \subseteq \Theta(m)$ space; Algorithm 6 (*forward*) achieves this.*

**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 $\Theta(n \log(n))$ 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 $\eta$. □

Compared to the two other time optimal algorithms we have presented, this algorithm is significantly more compact. Moreover, it is very simple and easy to implement, which also implies, as shown in [33,34], that it is very efficient in practice. In addition, it does not have the drawback of depending on a parameter $K$, central in Algorithm 5 (*ayz-listing*). Finally, we show in the next sections that it may be slightly modified to reduce further its space complexity (Section 4.3), and that even better performances can be proved if one considers power-law graphs (Section 6).

### 4.3. *Time-optimal* compact *algorithms for sparse graphs*

This section is devoted to time-optimal listing algorithms that have very low space requirements, both regarding the representation of $G$ and the additional space needed.

*A compact version of Algorithm 6 (*forward*)*

Thanks to the proof we gave of Theorem 10, it is now easy to modify Algorithm 6 (*forward*) in order to improve significantly its space complexity. This leads to the following result.

**Algorithm 7 – *compact-forward*.** *Lists all the triangles in a graph.*

```
Input: the adjacency array representation of $G$
Output: all the triangles in $G$
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 adjacency array 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 11.** *Given the adjacency array representation of G, it is possible to list all its triangles in $\Theta(m^{\frac{3}{2}})$ time and $\Theta'(2m+2n) \subseteq \Theta(m)$ space; Algorithm 7 (*compact-forward*) achieves this.*

**Proof.** Recall that, as explained in the proof of Theorem 10, when one computes the intersection of $A[v]$ and $A[u]$ (line 3aa of Algorithm 6 (*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 7 (*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 $O(n\log(n))$ time and $\Theta'(n)$ space cost. Moreover, sorting the simple compact representation of $G$ (line 2) is in $O(m\log(n))$ 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. □

Also note that we do not need to store the whole adjacency arrays representing $G$ in order to list the triangles using Algorithm 7 (*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 12.** *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 $\Theta(m\log(n))$ 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.

In practice, these results mean that one could encode vertices by integers, with the property that this numbering goes from highest degree vertices to lowest ones, then store the graph in the adjacency array representation, sort it, and compute the triangles using Algorithm 7 (*compact-forward*). In such a framework, and using the trick pointed out in the corollary above, the algorithm runs in $\Theta'(m)$ space, since line 1, responsible for the $\Theta'(n)$ cost, is unnecessary. On the other hand, if one wants to keep the original numbering of vertices, then one has to store the function $\eta()$ and renumber the vertices back after the triangle computation. This has an additional $\Theta'(n)$ space cost (and no significant time cost).

*A new algorithm*

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 we design an algorithm able to efficiently list the triangles of high degree vertices. Then, we use it in an algorithm similar to Algorithm 5 (*ayz-listing*), but that both avoids adjacency matrix representation, and reaches a $\Theta(m)$ space cost.

First note that we already have an algorithm listing all the triangles containing a given vertex $v$, namely Algorithm 2 (*vertex-listing*) [24]. This algorithm is in $\Theta(1)$ space (when the adjacency matrix is given), but it is inefficient on high degree vertices, since it needs $\Theta(d(v)^2)$ time. Our improved listing algorithm relies on an equivalent to Algorithm 2 (*vertex-listing*) that avoids this.

**Algorithm 8 – *new-vertex-listing.*** *Lists all the triangles containing a given vertex.*

```
Input: the adjacency array representation of $G$, and a vertex $v$
Output: all the triangles to which $v$ belongs
1. create an array $A$ of $n$ booleans and set them to false
2. for each vertex $u$ in $N(v)$, set $A[u]$ to true
3. for each vertex $u$ in $N(v)$:
  3a. for each vertex $w$ in $N(u)$:
    3aa. if $A[w]$ then output $\{v, u, w\}$
```

**Lemma 13.** *Given the adjacency array representation of G, it is possible to list all its triangles containing a given vertex $v$ in $\Theta(m)$ (optimal) time and $\Theta'(\frac{n}{\sigma}) \subseteq \Theta(n)$ space; Algorithm 8 (*new-vertex-listing*) achieves this.*

**Proof.** One may see Algorithm 8 (*new-vertex-listing*) as a way to use the adjacency matrix of $G$ without explicitly storing it: the array $A$ is nothing but the $v$-th line of the adjacency-matrix. It is constructed in $\Theta'(\frac{n}{\sigma})$ time and space (lines 1 and 2). [9] Then one can test for any edge $(v, u)$ in $\Theta(1)$ time and space. The loop starting at line 3 takes any edge containing one neighbor $u$ of $v$ and tests if its other end ($w$ in the algorithm) is linked to $v$ using $A$, in $\Theta(1)$ time and space. This is sufficient to find all the triangles containing $v$. Since this number of edges is bounded by $2m$ (one may actually obtain an equivalent algorithm by replacing lines 3a and 3aa by a loop over all the edges), we obtain that the algorithm is in $O(m)$ time and $\Theta'(\frac{n}{\sigma})$ space.

The obtained time complexity is optimal since $v$ may belong to $\Theta(m)$ triangles. □

**Algorithm 9 – *new-listing.*** *Lists all the triangles in a graph.*

```
Input: the sorted adjacency array representation of $G$, and an integer $K$
Output: all the triangles in $G$
1. for each vertex $v$ in $V$:
  1a. if $d(v) > K$ then, using Algorithm 8 (new-vertex-listing):
    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 $(u, v)$ using Algorithm 3 (edge-listing)
```

**Theorem 14.** *Given the sorted adjacency array representation of G, it is possible to list all its triangles in $\Theta(m^{\frac{3}{2}})$ time and $\Theta'(2m + n + \frac{n}{\sigma}) \subseteq \Theta(m)$ space; Algorithm 9 (*new-listing*) achieves this 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 13) (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 $O(mK)$ 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 $O(\frac{m}{K}m+mK)$ 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 11 and 14 improve Theorems 9 and 10 since they show that the same (optimal) time-complexity may be achieved in significantly less space.

Note however that it is still unknown whether there exist algorithms with time complexity in $\Theta(m^{\frac{3}{2}})$ but with lower space requirements. We saw that *edge-iterator* achieves $\Theta(md_{\max}) \subseteq O(mn)$ time with only a constant space requirement in addition to the adjacency array, and thus in $\Theta'(2m + n) \subseteq \Theta(m)$ space. In this direction, one may use the adjacency arrays to obtain the following stronger (if $d_{\max} \in \Omega(\sqrt{m}\log(n))$) result.

[9] Recall that $\sigma$ is the space needed to store an integer between 0 and $n$; here only one bit is necessary.

**Corollary 15.** *Given the adjacency array representation of G, it is possible to list all its triangles in $O(m^{\frac{3}{2}}\sqrt{\log(n)})$ time and $\Theta'(2m+n) \subseteq \Theta(m)$ space; Algorithm 9 (*new-listing*) achieves this if one takes $K \in \Theta(\sqrt{m\log(n)})$.*

**Proof.** Let us first sort the arrays in $O(m\log(n))$ time and $\Theta'(1)$ space. Then, we change Algorithm 8 (*new-vertex-listing*) by removing the use of $A$ and replace line 3aa by a dichotomic search for $w$ in $N(u)$, which has a cost in $O(\log(n))$ time and $\Theta'(1)$ space. Now if Algorithm 9 (*new-listing*) uses this modified version of Algorithm 8 (*new-vertex-listing*), then it is in $\Theta'(1)$ space and $O(\frac{m}{K}m\log(n)+mK)$ time. The optimal value for $K$ is then in $\Theta(\sqrt{m\log(n)})$, leading to the announced complexity. □

## 5. The case of power-law graphs

Until now, we presented several results which 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 also 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 huge 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 $\alpha$ generally between 2 and 3. See [36,13,1,29,37,19] for extensive lists of cases in which this property was observed. [10]

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 [18,15]. 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 the normalization constant. [11] 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 $p_k = \frac{1}{\alpha-1}\int_k^{k+1} x^{-\alpha}\mathrm{d}x = k^{-\alpha+1} - (k+1)^{-\alpha+1}$.

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 16.** *Given an adjacency array representation of a power-law graph G with exponent $\alpha$, Algorithm 7 (*compact-forward*) and Algorithm 9 (*new-listing*) with $K \in \Theta(n^{\frac{1}{\alpha}})$ list all its triangles in $O(mn^{\frac{1}{\alpha}})$ time and the same space complexities as above.*

**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 $\alpha$, 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 $n_K = nK^{-\alpha+1}$.

Let us first prove the result concerning Algorithm 9 (*new-listing*). As already noticed in the proof of Theorem 14, its space complexity does not depend on $K$. Moreover, its time complexity is in $O(n_K m + mK)$. The value of $K$ that minimizes this is in $\Theta(n^{\frac{1}{\alpha}})$, and the result for Algorithm 9 (*new-listing*) follows.

Let us now consider the case of Algorithm 7 (*compact-forward*). The space complexity was already proved for Theorem 11. The time complexity is the same as the one for Algorithm 6 (*forward*), and we use here the same notations as in the proof of Theorem 10. 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 6 (*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 of degree 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 6 (*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 $O(d(v) + n_{d(v)})$.

Like above, let us compute the value $K$ of $d(v)$ such that these two bounds are equal. We obtain $K = n^{\frac{1}{\alpha}}$. 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 6 (*forward*)), we obtain the announced complexity. □

Let us underline 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 $\alpha$ is large enough, and even if $m \in \Theta(n)$. This holds in particular for typical cases met in practice, where $\alpha$ often is between 2 and 3 [13,1]. It may be seen as an explanation of the

[10] Note that if $\alpha$ 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.

[11] One may also choose $p_k$ proportional to $\int_{k-\frac{1}{2}}^{k+\frac{1}{2}} x^{-\alpha}\mathrm{d}x$. Choosing any of this kind of solutions has little impact on the obtained results, see [15] and the proofs we present in this section.

fact that Algorithm 6 (*forward*) has very good performances on graphs with heterogeneous degree distributions, as shown experimentally in [33,34].

One may use the same kind of approach to prove better performances for Algorithm 1 (*ayz-node-counting*) and Algorithm 5 (*ayz-listing*) in the case of power-law graphs as follows.

**Corollary 17.** *Given the adjacency array representation of a power-law graph $G$ with exponent $\alpha$ and its adjacency matrix, it is possible to solve node-counting, counting and finding on $G$ in $O(n^{\frac{\omega\alpha+\omega}{\omega\alpha-\omega+2}})$ time and $\Theta(n^{\frac{2\alpha+2}{\omega\alpha-\omega+2}})$ space; Algorithm 1 (*ayz-node-counting*) achieves this if one takes $K$ in $\Theta(n^{\frac{\omega-1}{\omega\alpha-\omega+2}})$.*

**Proof.** With the same reasoning as the one in the proof of Theorem 2, one obtains that the algorithm runs in $O(nK^2 + (n_K)^\omega)$ where $n_K$ denotes the number of vertices of degree larger than $K$. As explained in the proof of Theorem 16, this is $n_K = nK^{-\alpha+1}$. Therefore, the best $K$ is such that $nK^2$ is in $\Theta(n^\omega K^{\omega(1-\alpha)})$. Finally, $K$ must be in $n^{\frac{1-\omega}{\omega(1-\alpha)-2}}$. One then obtains the announced time complexity. The space complexity is bounded by the space needed to construct the adjacency matrix between the vertices of degree at most $K$, thus it is $(n_K)^2$, and the result follows. □

If the degree distribution of $G$ follows a power law with exponent $\alpha = 2.5$ (typical for internet graphs [13,1]) then this result says that Algorithm 1 (*ayz-node-counting*) reaches a $O(n^{1.5})$ time and $O(n^{1.26})$ space complexity. If the exponent is larger, then the complexity is even better. Note that one may also obtain tighter bounds in terms of $m$ and $n$, for instance using the fact that Algorithm 1 (*ayz-node-counting*) has running time in $\Theta(mK + (n_K)^\omega)$ rather than $\Theta(nK^2 + (n_K)^\omega)$ (see the proofs of Theorem 2 and Corollary 17). We do not detail this here because the obtained results are quite technical and follow immediately from the ones we detailed.

**Corollary 18.** *Given the adjacency array representation of a power-law graph $G$ with exponent $\alpha$ and its adjacency matrix, it is possible to list all its triangles in $\Theta(mn^{\frac{1}{\alpha}})$ time and $\Theta(n^2)$ space; Algorithm 5 (*ayz-listing*) achieves this if one takes $K$ in $\Theta(n^{\frac{1}{\alpha}})$.*

**Proof.** The time complexity of Algorithm 5 (*ayz-listing*) is in $\Theta(mK + mn_K)$. The $K$ minimizing this is such that $K \in \Theta(n_K)$, which is the same condition as the one in the proof of Theorem 16; therefore we reach the same time complexity. The space complexity is bounded by the size of the adjacency matrix of $G'$ (with the same notation as in the proof of Theorem 9), i.e. $\Theta((n_K)^2)$. It is bounded by the size of the adjacency of $G$ itself, which leads to the announced complexity. □

Notice that this result implies that, for some reasonable values of $\alpha$ (namely $\alpha > 2$) the space needed *in addition to the graph representation* is in $o(n)$. This however is of theoretical interest only: it relies on the use of both the adjacency matrix and the adjacency array representation of $G$, which is unfeasible in practice for large graphs.

Finally, the results presented in this section show that one may use properties of most large graphs met in practice (here, their heterogeneous degree distribution), to improve results known on the general case (or on the sparse graph case).

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. One may therefore prove that they have even better performance (or that the bound is tight), and algorithms faster than the ones presented here may exist (for power-law graphs).

## 6. Experimental evaluation

In [33,34], the authors present a wide set of experiments on both real-world complex networks and some generated using various models, to evaluate experimentally the known algorithms. They focus on *vertex-iterator*, *edge-iterator*, Algorithm 6 (*forward*), and Algorithm 5 (*ayz-listing*), together with their counting and node-counting variants (they compute clustering coefficients). They also study variants of these algorithms using, for instance, hashtables and balanced trees. These variants have the same worst case asymptotic complexities but one may guess that they would run faster than the original algorithms, for several reasons we do not detail here. Matrix approaches are considered as too intricate to be used in practice.

The overall conclusion of their extensive experiments is that Algorithm 6 (*forward*) performs best on real-world (sparse and power-law) graphs: its asymptotic time is optimal and the constants involved in its implementation are very small. Variants, which need more subtle data structures, actually fail in performing better in most cases (because of the overhead induced by the management of these structures).

In order to integrate our contribution in this context and have a precise idea of the behavior of the discussed algorithms in practice, we also performed a wide set of experiments. We first discuss typical instances below, and detail a case for which previously known fast algorithms cannot be used because of space limitations.

We provide implementations at [26] which make it easy for anyone to compare the algorithms on his/her own instances (and to use them in applications).

**Table 1**
Performances of the main algorithms on typical real-world examples

| Graph | $n$ | $m$ | $d_{\max}$ | Time cost | | | | Space cost | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | el | f | cf | nl | el | f | cf | nl |
| Cooccurrences | 9 264 | 392 066 | 7 053 | 2 s | 0.25 s | 0.25 s | 0.25 s | 4 MB | 6 MB | 4 MB | 4 MB |
| Flickr groups | 75 121 | 88 650 430 | 43 720 | 2 h 26 min | 31 min | 31 min | 31 min | 700 MB | 1100 MB | 700 MB | 700 MB |
| Actor | 383 640 | 15 038 083 | 3 956 | 1 min 12 s | 20 s | 20 s | 28 s | 120 MB | 180 MB | 120 MB | 120 MB |
| IP exchanges | 467 273 | 1 744 214 | 81 756 | 6 s | 1 s | 1 s | 1 s | 16 MB | 26 MB | 17 MB | 16 MB |
| Notre-Dame | 701 654 | 1 935 518 | 5 331 | 1 s | 0.5 s | 0.5 s | 0.5 s | 18 MB | 31 MB | 21 MB | 18 MB |
| Flickr contacts | 1 920 914 | 10 097 185 | 30 167 | 68 s | 14 s | 14 s | 15 s | 85 MB | 138 MB | 92 MB | 85 MB |
| Phone calls | 2 527 730 | 6 340 925 | 1 230 | 3 s | 2 s | 2 s | 3 s | 59 MB | 102 MB | 68 MB | 61 MB |
| P2P exchanges | 6 235 399 | 159 870 973 | 15 420 | 29 min | 7 min | 7 min | 10 min | 1.3 GB | 1.9 GB | 1.3 GB | 1.3 GB |
| Webgraph | 39 459 925 | 783 027 125 | 1 776 858 | 41 h | – | 20 min | 45 min | 6 GB | 9.2 GB | 6.2 GB | 6.1 GB |

**From top to bottom:** a cooccurrence graph (nodes are the words in a book, and a link indicates that the two words appear in a same sentence); Flickr group graph (the nodes are `flickr` groups and a link indicates that the two groups have at least one member in common); an actor graph (the nodes are movie actors found in IMDB, with a link between two actors if they appear in a same movie); an exchange graph at IP level (nodes are IP addresses with a link between two addresses if the router on which the measurement is conducted routed a packet from one of them to the other); the Notre-Dame web graph (a graph in which nodes are the web pages at the university of Notre-Dame, with links between them); Flickr contact graph (nodes are `flickr` users, and there is a link between two users if one of them is a contact of the other); a phone call graph (nodes are phone numbers, with a link between two numbers if a call from one to the other occurred during the observation); a P2P exchange graph (nodes are the users of an *eDonkey* server, and two nodes are linked if they exchanged a file during the measurement); a web graph provided by the WebGraph project (pages in the `.uk` domain, with links between them, detailed below). **For each graph, we give, from left to right:** its number of nodes $n$, its number of links $m$, its maximal degree $d_{\max}$, the running times of Algorithm 3 (*edge-listing*) (el), Algorithm 6 (*forward*) (f), Algorithm 7 (*compact-forward*) (cf), and Algorithm 9 (*new-listing*) (nl) (with the best value for $K$, see below), and their respective space needs.

### *Typical real-world cases*

Table 1 shows the performances of the main algorithms discussed above in a variety of real-world cases. We do not give a detailed description of these graphs, which would be of little interest here; the key points are that these graphs span well the variety of cases met in practice, and they all have heterogeneous degree distributions well fitted by power-laws. Likewise, we do not give a detailed description of the machine running the experiments (a typical high performance workstation with a Dual Core AMD Opteron(tm) Processor 275 at 2.2 GHz and 8 GB of main memory), which would provide little useful information, if any; the running times are provided to help in comparing the different algorithms, not to predict the precise running time on a given computer.

Experimental results are in full accordance with theoretical predictions. Algorithm 3 (*edge-listing*) is very compact but may need much time to terminate, in particular in case of graphs with high-degree nodes. Algorithm 6 (*forward*) and Algorithm 7 (*compact-forward*) are very fast, but Algorithm 6 (*forward*) has a prohibitive space cost in some cases. Algorithm 7 (*compact-forward*) and Algorithm 9 (*new-listing*) are very compact (almost as much as Algorithm 3 (*edge-listing*), as predicted by the analysis), while being very fast. This makes them the most appealing solutions in practice.

One may notice that the computation times of all algorithms are strongly related to (though not a direct consequence of) the presence of high-degree nodes; this was a key point in all our formal analysis, which is confirmed by these experiments.

Note that Algorithm 9 (*new-listing*), just like Algorithm 1 (*ayz-node-counting*) and Algorithm 5 (*ayz-listing*), suffers from a serious drawback: it relies on the choice of a relevant value for $K$, the maximal degree above which vertices are considered as having a high degree. Though in theory this is not a problem, in practice it may be quite difficult to determine the best value for $K$, i.e. the one that minimizes the execution time. It depends both on the machine running the program and on the graph under concern.

We took here the best value, i.e. the one leading to the lowest execution time, in each case. We will discuss this in more details below. With this best value given, the time performances of Algorithm 9 (*new-listing*) are similar to, but lower than, the ones of Algorithm 6 (*forward*). Its space requirements are much lower, as predicted by Theorem 14. Likewise, Algorithm 9 (*new-listing*) speed is close to the one of Algorithm 7 (*compact-forward*) and it has slightly lower space requirements, in particular when the number of nodes is very large, as predicted.

### *A case previously our of reach*

It is important to note that the use of compact algorithms, namely Algorithm 7 (*compact-forward*) and Algorithm 9 (*new-listing*), makes it possible to manage graphs that were previously out of reach because of space requirements. To illustrate this, we detail now the experiment labelled *WebGraph* in Table 1 (last line), which previous algorithms were unable to manage in 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 [10]. 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* [11]. It has $n = 39\,459\,925$ vertices and $m = 783\,027\,125$ (undirected) edges, leading to more than 6 GB 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 Fig. 1, showing that the degrees are very heterogeneous (its maximal degree is $d_{\max} = 1\,776\,858$) and reasonably well fitted by a power-law of exponent $\alpha = 2.5$. It contains 304 529 576 triangles.

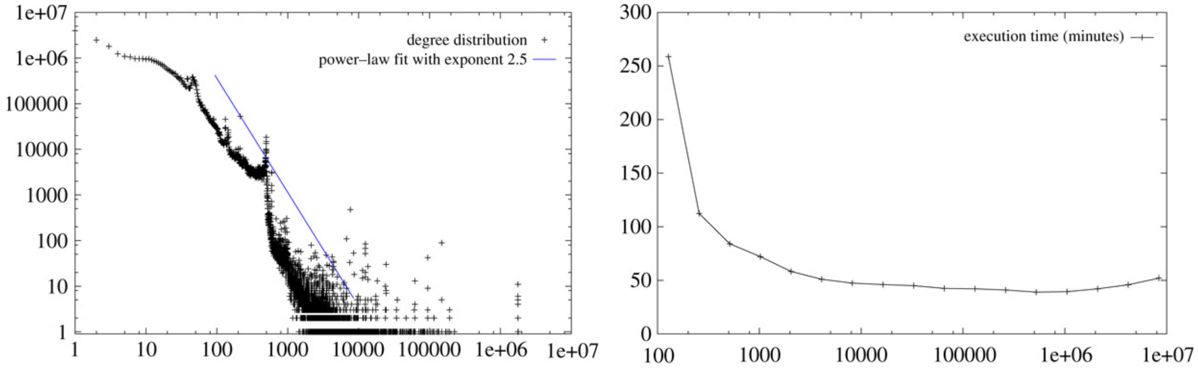

**Fig. 1.** Left: the degree distribution of our graph. Right: the execution time (in min) as a function of the number of vertices considered as high degree ones.

Let us emphasise that Algorithm 6 (*forward*), as well as the ones based on adjacency matrices, are unable to manage this graph on our 8 GB memory machine. Instead, and despite the fact that it is quite slow, *edge-iterator*, with its $\Theta'(2m+n) \subseteq \Theta(m)$ space complexity, can handle this. It took approximately 41 h to solve node-counting on this graph with this algorithm on our machine.

Algorithm 7 (*compact-forward*) achieves much better results: it took approximately 20 min. Likewise, Algorithm 9 (*new-listing*) took around 45 min (depending on the value of $K$). This is probably close to what Algorithm 6 (*forward*) would achieve if more main memory was available.

In order to use Algorithm 9 (*new-listing*) in such cases, one may evaluate the best $K$ in a preprocessing step at running time (by measuring the time needed to perform the key steps of the algorithm for various $K$). This can be done without changing the asymptotic complexity. However, there is a much simpler way to choose $K$, with negligible loss in performance, which we discuss now.

We plot in Fig. 1 (right) the running time of Algorithm 9 (*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. This may be seen as a consequence of the fact that *edge-iterator* is very efficient when the maximal degree is bounded; managing high degree vertices efficiently with Algorithm 8 (*new-vertex-listing*) and then the low degree ones with *edge-iterator* therefore leads to good performances. In other words, the few high degree vertices (which may be observed on the degree distribution plotted in Fig. 1) are responsible for the low performance of *edge-iterator*.

When $K$ decreases, the number of vertices with degree larger than $K$ increases, and the performances continue to be better and better for a while. They reach a minimal running time, and then the running time grows again. The other important point here is that this growth is very slow, and thus the performance of the algorithm remains close to its best for a wide range of values of $K$. This implies that, with any reasonable guess for $K$, the algorithm performs well.

## 7. Conclusion

In this contribution, we gave a detailed survey of existing results on triangle problems, and we completed them in two directions. First, we gave the space complexity of each previously known algorithm. Second, we proposed new algorithms that achieve both optimal time complexity and low space needs. Taking space requirements into account is a key issue in this context, since this currently is the bottleneck for triangle problems when the considered graphs are very large. This is discussed on a practical case in Section 6, where we show that our compact algorithms make it possible to handle cases that were previously out of reach.

Another significant contribution of this paper is the analysis of algorithm performances on power-law graphs (Section 5), which model a wide variety of very large graphs met in practice. We were able to show that, on such graphs, several algorithms have better performance than in the general (sparse) case.

Finally, the current state of the art concerning triangle problems, including our new results, may be summarized as follows:

- except the fact that node-counting may have a $\Theta(n)$ space overhead (depending on the underlying algorithm), there is no known difference in time and space complexities between finding, counting, and node-counting;
- the fastest known algorithms for these three problems rely on matrix product and are in $O(n^{2.376})$ or $O(m^{1.41})$ time and $\Theta(n^2)$ space ( Theorems 1 and 2); however, no lower bound better than the trivial $\Omega(m)$ one is known for the time complexity of these problems;
- the other known algorithms rely on solutions to the listing problem and have the same performances as on this problem; they are slower than matrix approaches but need less space;
- listing can be solved in $\Theta(n^3)$ or $\Theta(nm)$ (optimal in the general case) time and $\Theta(n^2)$ or $\Theta(m)$ (optimal) space ( Theorems 4–6) ; this can be achieved from the sorted adjacency array representation of the graph;

- listing may also be solved in $\Theta(m^{\frac{3}{2}})$ (optimal in the general and sparse cases) time and $\Theta(m)$ space ( Theorems 11 and 14), still from the adjacency array representation of the graph; this is much better for sparse graphs;
- if main memory is very limited, one may use Corollary 12 to solve triangle listing in $\Theta'(m+n) \subseteq \Theta(m)$, while keeping the optimal $\Theta(m^{\frac{3}{2}})$ time complexity; using external memory, this may even be reduced to $\Theta'(m) \subseteq \Theta(m)$ main memory needs, as discussed at the end of Section 4.3;
- in the case of power-law graphs, it is possible to prove better complexities, leading to $O(mn^{\frac{1}{\alpha}})$ time and compact solutions (where $\alpha$ is the exponent of the power-law) (Theorem 16);
- in practice, it is possible to obtain very good performances (both concerning time and space needs) using Algorithm 7 (*compact-forward*) and Algorithm 9 (*new-listing*).

We detailed several other results, but they are weaker (they need the adjacency matrix of the graph in input and/or have higher complexities) than these ones.
This contribution also opens several questions for further research, most of them related to the tradeoff between space and time efficiency. Let us cite for instance:

- can matrix approaches be modified in order to induce lower space complexity?
- is listing feasible in less space, while still in optimal time $\Theta(m^{\frac{3}{2}})$?
- is it possible to design a listing algorithm with complexity $o(mn^{\frac{1}{\alpha}})$ time and $o(m)$ space for power-law graphs with exponent $\alpha$? what is the optimal time complexity in this case?

It is also important to notice that other approaches exist, based for instance on streaming algorithmics (avoiding to store the graph in main memory) [23,4,25] and/or approximate algorithms [32,25,35], and/or various methods to compress the graph [8,9]. These approaches are very promising for graphs even larger than the ones considered here, in particular the ones that do not fit in main 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 algorithms if the graph contains few triangles. Likewise, it is reasonable to expect that triangle listing, but also node-counting and counting, may perform poorly if there are many triangles in the graph. The finding problem, on the contrary, 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 5 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. For instance, results on planar graphs are provided in [24], and results using arboricity in [14,3]. 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. I also thank Paolo Boldi from the WebGraph project [10], who provided the data used in Section 6. This work was partly funded by the MetroSec (Metrology of the Internet for Security) [40] and PERSI (Programme d'Étude des Réseaux Sociaux de l'Internet) [41] projects.

## References

[1] R. Albert, A.-L. Barabási, Statistical mechanics of complex networks, Rev. Modern Phys. 74 (47) (2002).
[2] Noga Alon, Raphael Yuster, Uri Zwick, Finding and counting given length cycles, in: European Symposium Algorithms, ESA, 1994.
[3] Noga Alon, Raphael Yuster, Uri Zwick, Finding and counting given length cycles, Algorithmica 17 (3) (1997) 209–223.
[4] Ziv Bar-Yossef, Ravi Kumar, D. Sivakumar, Reduction in streaming algorithms with an application of counting triangles in graphs, in: ACM/SIAM Symposium On Discrete Algorithms, SODA, 2002.
[5] Vladimir Batagelj, Personnal communication, 2006.
[6] Vladimir Batagelj, Andrej Mrvar, Pajek: A program for large network analysis, Connections 21 (2) (1998) 47–57.
[7] Vladimir Batagelj, Andrej Mrvar, A subquadratic triad census algorithm for large sparse networks with small maximum degree, Social Networks 23 (2001) 237–243.
[8] P. Boldi, S. Vigna, The webgraph framework i: Compression techniques, in: WWW, 2004.
[9] P. Boldi, S. Vigna, The webgraph framework ii: Codes for the world-wide web, in: DCC, 2004.
[10] Paolo Boldi, WebGraph project, http://webgraph.dsi.unimi.it/.
[11] Paolo Boldi, Bruno Codenotti, Massimo Santini, Sebastiano Vigna, Ubicrawler: A scalable fully distributed web crawler, Softw., Pract. Exper. 34 (8) (2004) 711–726.
[12] Béla Bollobás, Oliver M. Riordan, Mathematical results on scale-free random graphs, in: Handbook of Graphs and Networks: From the Genome to the Internet, Wiley-VCH, 2002.
[13] U. Brandes, T. Erlebach (Eds.), Network Analysis: Methodological Foundations, in: LNCS, Springer-Verlag, 2005.
[14] Norishige Chiba, Takao Nishizeki, Arboricity and subgraph listing algorithms, SIAM J. Comput. 14 (1985).
[15] R. Cohen, R. Erez, D. ben Avraham, S. Havlin, Reply to the comment on 'breakdown of the internet under intentional attack', Phys. Rev. Lett. 87 (2001).
[16] Don Coppersmith, Shmuel Winograd, Matrix multiplication via arithmetic progressions, J. Symb. Comput. 9 (3) (1990) 251–280.
[17] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, Introduction to Algorithms, second edition, MIT Press, 2001.

[18] S.N. Dorogovtsev, J.F.F. Mendes, Comment on 'breakdown of the internet under intentional attack', Phys. Rev. Lett. 87 (2001).

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

[20] Gianni Franceschini, S. Muthukrishnan, Mihai Pătraşcu, Radix sorting with no extra space, in: European Symposium on Algorithms, ESA, 2007, pp. 194–205.

[21] Frank Harary, Helene J. Kommel, Matrix measures for transitivity and balance, J. Math. Sociology (1979).

[22] Frank Harary, Herbert H. Paper, Toward a general calculus of phonemic distribution, Language: J. Linguistic Soc. America 33 (1957) 143–169.

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

[24] Alon Itai, Michael Rodeh, Finding a minimum circuit in a graph, SIAM J. Comput. 7 (4) (1978) 413–423.

[25] H. Jowhari, M. Ghodsi, New streaming algorithms for counting triangles in graphs, in: COCOON, 2005.

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

[27] Matthieu Latapy, Clémence Magnien, Complex network measurements: Estimating the relevance of observed properties, in: Proceedings of Infocom'08, Phoenix, USA, 2008.

[28] J. Leskovec, J. Kleinberg, C. Faloutsos, Graphs over time: Densification laws, shrinking diameters and possible explanations, in: ACM SIGKDD, 2005.

[29] Ron Milo, Shai Shen-Orr, Shalev Itzkovitz, Nadav Kashtan, Dmitri Chklovskii, Uri Alon, Network motifs: Simple building blocks of complex networks, Science 298 (2002) 824–827.

[30] Burkhard Monien, How to find long paths efficiently, Ann. Discrete Math. 25 (1985) 239–254.

[31] C.R. Edling, P. Holme, F. Liljeros, Structure and time-evolution of an internet dating community, Social Networks 26 (2) (2004).

[32] Thomas Schank, Dorothea Wagner, Approximating clustering coefficient and transitivity, J. Graph Algorithms Appl. (JGAA) 9 (2) (2005) 265–275.

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

[34] Thomas Schank, Dorothea Wagner, Finding, counting and listing all triangles in large graphs, an experimental study, in: Workshop on Experimental and Efficient Algorithms, WEA, 2005.

[35] Asaf Shapira, Noga Alon, Homomorphisms in graph property testing — a survey, in: Electronic Colloquium on Computational Complexity, ECCC, 2005.

[36] Duncan J. Watts, Steven H. Strogatz, Collective dynamics of smallworld networks, Nature 393 (1998) 440–442.

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

[38] Raphael Yuster, Uri Zwick, Detecting short directed cycles using rectangular matrix multiplication and dynamic programming, in: ACM/SIAM Symposium on Discrete Algorithms, SODA, 2004, pp. 254–260.

[39] Raphael Yuster, Uri Zwick, Fast sparse matrix multiplication, in: European Symposium on Algorithms, ESA, 2004, pp. 604–615.

[40] Metrosec project, http://www2.laas.fr/METROSEC/.

[41] Persi project, http://www.liafa.jussieu.fr/~persi/.