Tailored vertex ordering for faster triangle listing in large graphs

Fabrice Lécuyer
fabrice.lecuyer@lip6.fr
Sorbonne Université, CNRS, LIP6
F-75005 Paris, France

Louis Jachiet
Télécom Paris
Institut Polytechnique de Paris

Clémence Magnien
Sorbonne Université, CNRS, LIP6
F-75005 Paris, France

Lionel Tabourier
Sorbonne Université, CNRS, LIP6
F-75005 Paris, France

ABSTRACT

Listing triangles is a fundamental graph problem with many applications, and large graphs require fast algorithms. Vertex ordering allows the orientation of edges from lower to higher vertex indices, and state-of-the-art triangle listing algorithms use this to accelerate their execution and to bound their time complexity. Yet, only two basic orderings have been tested. In this paper, we show that studying the precise cost of algorithms instead of their bounded complexity leads to faster solutions. We introduce cost functions that link ordering properties with the running time of a given algorithm. We prove that their minimization is NP-hard and propose heuristics to obtain new orderings with different trade-offs between cost reduction and ordering time. Using datasets with up to two billion edges, we show that our heuristics accelerate the listing of triangles by an average of 30% when the ordering is already given as an input, and 15% when the ordering time is included.

CCS CONCEPTS

• **Theory of computation → Design and analysis of algorithms**; **Graph algorithms analysis**; • **Mathematics of computing → Graph algorithms**; • **Information systems → Data mining**.

KEYWORDS

graph algorithm, pattern mining, scalability, vertex ordering, real-world network

ACM Reference Format:
Fabrice Lécuyer, Louis Jachiet, Clémence Magnien, and Lionel Tabourier. 2022. Tailored vertex ordering for faster triangle listing in large graphs. In *Proceedings of conference title from your rights confirmation email (Conf 'XX).* ACM, New York, NY, USA, 7 pages. https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

1.1 Context

Small connected subgraphs are key to identifying families of real-world networks [21] and are used for descriptive or predictive purposes in various fields such as biology [23, 28], sociology [7, 9] and engineering [30]. In particular, listing elementary patterns such as triangles is a stepping stone to analyzing the structure of networks and their evolution [17, 27]. This task may seem simple, but web crawlers and social platforms generate graphs that are so large that scalability becomes a challenge. Thus, a lot of effort has been dedicated to efficient in-memory triangle listing. Note that streaming methods exist for graphs that do not fit in main memory [4, 10] and that exact or approximate methods designed for triangle *counting* [3, 13, 31] can generally not be adapted to triangle *listing*.

An efficient algorithm for triangle listing has been proposed early on in [8]. Based on the observation that real-world graphs generally have a heterogeneous degree distribution, later contributions [15, 26] showed how ordering vertices by degree or core value accelerates the listing. Since then, a unifying description has been proposed [22] and the method has been successfully extended to larger cliques [11, 19, 29]. However, only degree and core orderings have been exploited, but their properties are not specifically tailored for the triangle listing problem. Other types of orderings benefited other problems such as graph compression [6, 12] or cache optimization [16, 32]. The main purpose of this work is thus to find a general method to design efficient vertex orderings for triangle listing.

1.2 Contributions

In this work, we show how vertex ordering directly impacts the running time of the two fastest existing triangle listing algorithms. First, we introduce cost functions that relate the vertex ordering and the running time. We prove that finding an optimal ordering to minimize them is NP-hard. Then, we expose a gap in the combinations of algorithm and ordering considered in the literature, and we bridge it with three heuristics to reduce the corresponding costs. Our heuristics reach a compromise between their running time and the quality of the obtained ordering, in order to address two distinct tasks: listing triangles with or without taking into account the ordering time. Finally, we show that our resulting combinations of algorithm and ordering outperform state-of-the-art running times for either task. We release an efficient open-source implementation of all considered methods, available at [2].

Section 2 presents state-of-the-art methods to list triangles. In Section 3, we analyze the cost induced by a given ordering on these algorithms and propose several heuristics to reduce it; the proofs of NP-hardness are in appendix. The experiments of Section 4 show that our methods are efficient in practice and improve the state of the art.

Conf 'XX, May 02–03, 2022, City, Country

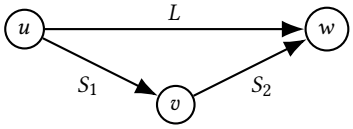


Figure 1: Directed triangle with the unified notations proposed in [22]. The edges are directed according to an ordering π such that $\pi_u < \pi_v < \pi_w$.

1.3 Notations

We consider an unweighted undirected simple graph $G = (V, E)$ with $n = |V|$ vertices and $m = |E|$ edges. The set of neighbors of a vertex u is denoted $N_u = \{v, \{u, v\} \in E\}$, and its degree is $d_u = |N_u|$. An ordering π is a permutation over the vertices that gives a distinct index $\pi_u \in [\![1, n]\!]$ to each vertex u. In the directed acyclic graph (DAG) $G_\pi = (V, E_\pi)$, for $\{u, v\} \in E$, E_π contains (u, v) if $\pi_u < \pi_v$, and (v, u) otherwise. In such a directed graph, the set N_u of neighbors of u is partitioned into its predecessors N_u^- and successors N_u^+. We define the indegree $d_u^- = |N_u^-|$ and the outdegree $d_u^+ = |N_u^+|$; their sum is $d_u^- + d_u^+ = d_u$. A triangle of G is a set of vertices $\{u, v, w\}$ such that $\{u, v\}, \{v, w\}, \{u, w\} \in E$. A k-clique is a set of k fully-connected vertices. The coreness c_u of vertex u is the highest value k such that u belongs to a subgraph of G where all vertices have degree at least k; the core value or degeneracy $c(G)$ of G is the maximal c_u for $u \in V$. A core ordering π verifies $\pi_u \leq \pi_v \Leftrightarrow c_u \leq c_v$. Core value and core ordering can be computed in linear time [5].

2 STATE OF THE ART

2.1 Triangle listing algorithms

Ortmann and Brandes [22] have identified two families of triangle listing algorithms: *adjacency testing*, and *neighborhood intersection*. The former sequentially considers each vertex u as a seed, and processes all pairs (v, w) of its neighbors; if they are themselves adjacent, $\{u, v, w\}$ is a triangle. Algorithms `tree-lister` [14], `node-iterator` [26] and `forward` [26] belong to this category. In contrast, the neighborhood intersection family methods sequentially considers each edge (u, v) as a seed; each common neighbor w of u and v forms a triangle $\{u, v, w\}$. Algorithms `edge-iterator` [26], `compact-forward` [15] and `K3` [8] belong to this category, as well as some algorithms that list larger cliques [11, 19, 20].

In naive versions of both adjacency testing and neighborhood intersection, finding a triangle (u, v, w) does not prevent from finding triangle (v, w, u) at a later step. The above papers avoid this unwanted redundancy by using an ordering, explicitly or not. We use the framework developed in [22]: a total order π is defined over the vertices, and the triple (u, v, w) is only considered a valid triangle if $\pi_u < \pi_v < \pi_w$. This guarantees that each triangle is listed only once: as illustrated in Figure 1, vertices in any triangle of the DAG G_π appear in one and only one of three positions: u is first, v is second, w is third; the same holds for edges: L is the long edge, and S_1 and S_2 are the first and second short edges. It leads to three variants of adjacency testing (seed vertex v or w instead of u) and of neighborhood intersection (seed edge L or S_2 instead of S_1).

Choosing the right data-structure is key to the performance of algorithms. All triangle listing algorithms have to visit the neighborhoods of vertices. Using hash table or binary tree to store them is very effective: they respectively allow for constant and logarithmic search on average. However, because of high constants, they are reportedly slow in terms of actual running time [26]. A faster structure is the boolean array used in `K3` for neighborhood intersection. It registers the elements of N_u^+ in a boolean table B so that, for each neighbor v of u, it is possible to check in constant time if a neighbor w of v is also a neighbor of u. This is the structure used by the fastest methods [11, 22].

In the rest of this paper, we therefore only consider triangle listing algorithms that use neighborhood intersection and a boolean array. We present the two that we will study in Figures 2 and 3 with the notations of Figure 1 for the vertices [1]. They initialize the boolean array B to false (line 1), consider a first vertex (line 2) and store its neighbors in B (line 3); then, for each of its neighbors (line 4), they check if their neighbors (line 5) are in B (line 6), in which case the three vertices form a triangle (line 7). B is reset (line 8) before continuing with the next vertex. The algorithm of Figure 2 corresponds to `L+n` in [22]; we call it `A++` because of the two "+" involved in its complexity described in Property 1. The algorithm of Figure 3 corresponds to $\mathtt{S}_1$`+n` in [22]; we call it `A+-`. Their complexities are given in Property 1. Since they depend on the indegree and outdegree of vertices, the choice of ordering will impact the running time of the algorithms.

Property 1 (Complexity of `A++` and `A+-`). *The complexity of* `A++` *is* $\Theta(\sum_{u \in V} d_u^{+2})$. *The complexity of* `A+-` *is* $\Theta\left(m + \sum_{v \in V} d_v^+ d_v^-\right)$.

Proof. In both algorithms, the boolean table B requires n initial values, m set and m reset operations, which is $\Theta(m)$ assuming that $n \in O(m)$. In `A++`, a given vertex u appears in the loop of Line 4 as many times as it has a successor w; every time, a loop over each of its successors v is performed. In total, u is involved in $\Theta(d_u^{+2})$ operations. Similarly, in `A+-`, a given vertex v appears in the loop of Line 4 as many times as it has a predecessor u; every time, a loop over each of its successors w is performed. In total, v is involved in $\Theta(d_v^+ d_v^-)$ operations. The term m is omitted in the complexity of `A++` as $\sum_{u \in V} d_u^{+2} \geq \sum_{u \in V} d_u^+ = m$, but not in `A+-` as $\sum_{v \in V} d_v^+ d_v^-$ can be lower than m. □

2.2 Orderings and complexity bounds

Ortmann and Brandes [22] order the vertices by non-decreasing degree or core value. In their experimental comparison, they test all the above methods with degree ordering, core ordering, and with the original ordering of the dataset. They conclude that the fastest method is `A++` with core or degree ordering: core is faster to list triangles when the ordering is given as an input, and degree is faster when the time to compute the ordering is also included.

Danisch *et al.* [11] also use core ordering in the more general problem of listing k-cliques. For triangles ($k = 3$), their algorithm is equivalent to `A+-`, and they show that using core ordering outperforms the methods of [8, 15, 20].

[1] Note that a third natural variant exists, which would be `A--` (or $\mathtt{S}_2$`+n`), but we do not consider it here since its complexity is equivalent to the one of `A++`.

Algorithm A++

```
for each vertex v do B[v] ← False
for each vertex w do
    for v ∈ N_w^- do B[v] ← True
    for u ∈ N_w^- do
        for v ∈ N_u^+ do
            if B[v] then
                output triangle{u, v, w}
    for v ∈ N_w^- do B[v] ← False
```

Complexity:

$$\Theta\Big(m + \sum_{(u,w)\in E_\pi} d_u^+\Big) = \Theta\Big(\sum_{u\in V} {d_u^+}^2\Big)$$

Figure 2: Algorithm A++ (or L+n)

Algorithm A+−

```
for each vertex w do B[w] ← False
for each vertex u do
    for w ∈ N_u^+ do B[w] ← True
    for v ∈ N_u^+ do
        for w ∈ N_v^+ do
            if B[w] then
                output triangle{u, v, w}
    for w ∈ N_u^+ do B[w] ← False
```

Complexity:

$$\Theta\Big(m + \sum_{(u,v)\in E_\pi} d_v^+\Big) = \Theta\Big(m + \sum_{v\in V} d_v^+ d_v^-\Big)$$

Figure 3: Algorithm A+− (or S_1+n)

With these two orderings, it is possible to obtain upper-bounds for the time complexity in terms of graph properties. Chiba and Nishizeki [8] show that K3 with degree ordering has a complexity in $O(m \cdot \alpha(G))$, where α is the arboricity. With core ordering, node-iterator-core [26] and kClist [11] have complexity $O(m \cdot c(G))$, where c is the core value. These bounds are considered equal in [22], following the proof in [33] that $\alpha(G) \leq c(G) \leq 2\alpha(G) - 1$. However, we focus in this work on the complexities expressed in Algorithms 2 and 3 as we will see that they describe the running time more accurately.

3 NEW ORDERINGS TO REDUCE THE COST OF TRIANGLE LISTING

3.1 Formalizing the cost of triangle listing algorithms

In this section, we discuss how to design vertex orderings to reduce the cost of triangle listing algorithms. For this purpose, we introduce the following costs that appear in the complexity formulas of Algorithms 2 and 3. Recall that the initial graph is undirected and that the orientation of edges is given by the ordering π, which partitions neighbors into successors and predecessors.

Definition 1 (Cost induced by an ordering). *Given an undirected graph G, the costs C^{++} and C^{+-} induced by a vertex ordering π are defined by:*

$$C^{++}(\pi) = \sum_{u\in V} d_u^+ d_u^+ \qquad C^{+-}(\pi) = \sum_{u\in V} d_u^+ d_u^-$$

The fastest methods in the state of the art are A++ with core or degree ordering [22], and A+− with core ordering [11]. The intuition of both orderings is that high degree vertices are ranked after most of their neighbors in π so that their outdegree in G_π is lower. This reduces the cost C^{++}, which in turn reduces the number of operations required to list all the triangles as well as the actual running time of A++. In [22], it is mentioned that core ordering performs well with A+− as a side effect.

To our knowledge, no previous work has designed orderings with a low C^{+-} cost and used them with A+−. We will show that such orderings can lower the computational cost further. Yet, optimizing C^{+-} or C^{++} is computationally hard because of Theorem 1:

Theorem 1 (NP-hardness). *Given a graph G, it is NP-hard to find an ordering π that minimizes $C^{+-}(\pi)$ or that minimizes $C^{++}(\pi)$.*

The result for C^{+-} was already known [25] but, as far as we know, no proof has been published. In an online supplementary material [1], we give a new simpler proof for the hardness of C^{+-}, and we prove the result for C^{++}.

3.2 Distinguishing two tasks for triangle listing

Triangle listing typically consists of the following steps: loading a graph, computing a vertex ordering, and listing the triangles. Time measurements in [11, 15, 19] only take the last step into account, while [22, 26] also include the other steps. We therefore address two distinct tasks in our study: we call **mere-listing** the task of listing the triangles of an already loaded graph with a given vertex ordering; we call **full-listing** the task of loading a graph, computing a vertex ordering, and listing its triangles.

In the rest of the paper, we use the notation *task-order-algorithm*: for instance, mere-core-A+− refers to the mere-listing task with core ordering and algorithm A+−. Using this notation, the fastest methods identified in the literature are mere-core-A+− in [11], mere-core-A++ and full-degree-A++ in [22].

With mere-listing, the ordering time is not counted, which allows to spend a long time to find an ordering with low cost. On the other hand, full-listing favors quickly obtained orderings even if their induced cost is not the lowest. For this reason, there is a time-quality trade-off for cost-reducing heuristics.

3.3 Reducing C^{+-} along a time-quality trade-off

The goal here is to design an algorithm that takes a graph as input and produces an ordering π with a low induced cost $C^{+-}(\pi)$. Because of Theorem 1, finding an optimal solution is not realistic for graphs with millions of edges. We therefore present three heuristics aiming at reducing the C^{+-} value, exploring the trade-off between quality in terms of C^{+-} and ordering time.

3.3.1 Neigh heuristic. We define the *neighborhood optimization* method, a greedy reordering where each vertex is placed at the optimal index with respect to its neighbors, as illustrated in Figure 5. First notice that changing an index π_u only affects $C^{+-}(\pi)$ if the position of u with respect to at least one of its neighbors changes; otherwise the in- and outdegrees of all vertices remain unchanged.

Algorithm Neigh
Input: graph G, initial ordering π, threshold $\epsilon \geq 0$
1: **repeat**
2: $C_0 = C^{+-}(\pi)$
3: **for** each vertex u of G **do**
4: sort N_u according to π
5: p_0 = position of u in its neighborhood
6: $p_* = \text{argmin}_p \{C^{+-}(p)\}$
7: update ordering π to put u in position p_*
8: **while** $C^{+-}(\pi) < (1-\epsilon) \cdot C_0$

Figure 4: Neighborhood optimization (*Neigh* heuristic)

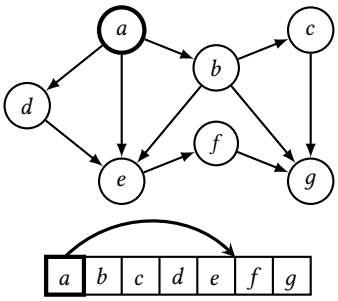
Starting from any ordering π, Algorithm 4 considers each vertex u one by one (line 3) and, for each $p \in [\![0, d_u+1]\!]$, it computes $C^{+-}(p)$, the value of C^{+-} when u is just before its p-th neighbor in π. The position p_* that induces the lowest value of C^{+-} is selected (line 6) and the ordering is updated (line 7). The process is repeated until C^{+-} reaches a local minimum, or until the relative improvement is under a threshold ϵ (last line). The resulting π induces a low C^{+-} cost.

For a vertex u, sorting the neighborhood according to π takes $O(d_u \log d_u)$ operations; finding the best position takes $\Theta(d_u)$ because it only depends on the values d_v^+ and d_v^- of each neighbor v of u. With a linked list, π is updated in constant time. If Δ is the highest degree in the graph, one iteration over all the vertices thus takes $O(m \log \Delta)$, which leads to a total complexity $O(Im \log \Delta)$ if the improvement threshold ϵ is reached after I iterations. Notice that on all the tested datasets the process reaches $\epsilon = 10^{-2}$ after less than ten iterations.

This heuristic has several strong points: it can be used for other objective functions, for instance C^{++}; because it is greedy, the cost keeps improving until the process stops; if the initial ordering already induces a low C^{+-} cost, the heuristic can only improve it; it is stable in practice, which means that starting from several random orderings give similar final costs; and we show in Section 4 that it allows for the fastest mere-listing.

In spite of its log-linear complexity, this heuristic can take longer than the listing of triangles itself in practice, which is an issue for the full-listing task. We therefore propose the following faster heuristics in the case of full-listing.

3.3.2 Check heuristic. This heuristic is inspired by core ordering, where vertices are repeatedly selected according to their current degree [5]. It considers all vertices by decreasing degree and *checks* if it is better to put a vertex at the beginning or at the end of the ordering. More specifically, π is obtained as follows: before placing vertex u, let V_b (resp. V_e) be the vertices that have been placed at the beginning (resp. at the end) of the ordering, and $V_?$ those that are yet to place. The neighbors of u are partitioned in $N_b = N_u \cap V_b$, $N_e = N_u \cap V_e$ and $N_? = N_u \cap V_?$. We consider two options to place u: either just after the vertices in V_b ($\pi_u = |V_b| + 1$), or just before the vertices in V_e ($\pi_u = n - |V_e|$). In either case, u has all vertices of N_b as predecessors, and all vertices of N_e as successors. In the first case, vertices in $N_?$ become successors, which induces a C^{+-} cost $C_b = |N_b| \cdot (|N_e| + |N_?|)$. In the second, the cost is



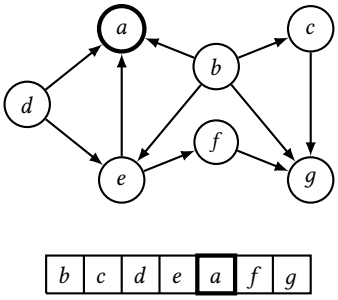


Figure 5: Example of update in the *Neigh* heuristic: vertex a is moved to a position among its neighbors that induces the lowest cost. The tables indicates how the ordering is updated. The edge in the DAG are reoriented accordingly. Here, the ordering at the top has $C^{+-} = 9$ while the ordering at the bottom has $C^{+-} = 6$. For this graph, the optimal cost is 3 (with ordering b, c, d, a, f, e, g).

$C_e = (|N_b| + |N_?|) \cdot |N_e|$. The option with the smaller cost is selected. Sorting the vertices by degree requires $O(n)$ steps with bucket sort. Maintaining the sizes of N_b, N_e, $N_?$ for each vertex requires one update for each edge. Therefore, the complexity is $O(m+n)$, or $O(m)$ assuming that $n \in O(m)$.

3.3.3 Split heuristic. Finally, we propose a heuristic that is faster but compromises on the quality of the resulting ordering. Degree ordering has been identified as the best solution for mere-listing with algorithm-A++ [22]. We adapt it for C^{+-} by *splitting* vertices alternatively at the beginning and at the end of the ordering π. More precisely, a non-increasing degree ordering δ is computed, then the vertices are split according to their parity: if u has index $\delta_u = 2i+1$ then $\pi_u = i+1$; if $\delta_u = 2i$, then $\pi_u = n+1-i$. Thus, high degree vertices will have either few predecessors or few successors, which ensures a low C^{+-}cost. The complexity of this method is in $O(n)$ like the degree ordering.

4 EXPERIMENTS

4.1 Experimental setup

4.1.1 Datasets. We use the 12 real-world graphs described in Table 1. Loops have been removed and the directed graphs have been transformed into undirected graphs by keeping one edge when one existed in either or both directions.

Table 1: Datasets used for the experiments, ranked by number of edges. They represent either web networks ★, social networks ▲ or citation networks ■.

dataset [source]	vertices	edges	triangles
skitter ★[18]	1,696,415	11,095,298	28,769,868
patents ■[18]	3,774,768	16,518,947	7,515,023
baidu-baike ★[24]	2,141,301	17,014,946	25,207,196
pokec ▲[18]	1,632,804	22,301,964	32,557,458
socfba-anon ▲[24]	3,097,166	23,667,394	55,606,428
LiveJournal ▲[18]	4,036,538	34,681,189	177,820,130
wiki ★[18]	2,070,486	42,336,692	145,707,846
orkut ▲[18]	3,072,627	117,185,083	627,584,181
it-2004 ★[6]	41,291,318	1,027,474,947	48,374,551,054
twitter-2010 ▲[6]	41,652,230	1,202,513,046	34,824,916,864
friendster ▲[18]	124,836,180	1,806,067,135	4,173,724,142
sk-2005 ★[6]	50,636,151	1,810,063,330	84,907,041,475

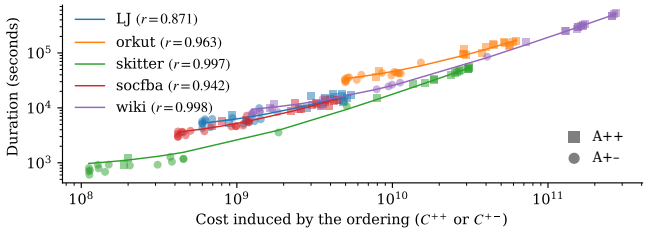


Figure 6: Running time of algorithms with respect to the cost induced by the ordering. Each mark represents an ordering: circles are for cost C^{+-} and algorithm A+-, squares are for cost C^{++} and algorithm A++. Each color represents a dataset: the line of linear regressions and the associated correlation coefficients r show the proportionality between cost and time.

4.1.2 Software and hardware. We release a uniform open-source implementation [2] of A++ and A+- algorithms, as well as the different ordering strategies that we discussed. Our implementation allows to run either algorithm in parallel, which is possible because each iteration of the main loop is independent from the others. Among orderings however, only degree and *Split* are easily parallelizable; to be consistent, we use a single thread to compare the different methods. The code is in `c++` and uses `gnu make 4` and the compiler `g++ 8.2` with optimization flag `Ofast` and `openmp` for parallelisation. We run all the programs on a `sgi ub2000 intel xeon e5-4650L @2.6 GHz, 128Gb ram` running `linux suse 12.3`.

Regarding the state of the art, the most competitive implementation available is `kClist` in `c` [11], which has already been shown to outperform previous programs [15, 20]. It lists k-cliques using a core ordering and a recursive algorithm that is equivalent to A+- for $k = 3$. We compared our implementation to `kClist` with various performance tests and found that ours is 14% faster on average, presumably because it does not use recursion. Therefore, we only use our own implementation for A+- in the rest of this paper, to ensure that the results depend on the combination of ordering and algorithm, but not on programming differences.

4.2 Cost and running time are linearly correlated

In order to show that the cost functions C^{++} and C^{+-} are good estimates of the running time, we measure the correlation between the running time of mere-listing and the corresponding cost induced by various orderings (core, degree, our heuristics, but also breadth- and depth-first search, random ordering, etc). In Figure 6, we see that the running time for a given dataset correlates almost linearly to the corresponding cost: the lines represent linear regressions. It only presents some of the datasets for readability, but the correlation is above 0.82 on all of them. In other words, the execution time of a listing algorithm is almost a linear function of the cost induced by the ordering, which is why reducing this cost actually improves the running time, as we will see.

4.3 *Neigh* outperforms previous mere-listing methods.

We compare our methods to the state of the art for mere-listing (core-A+- in [11] and core-A++ in [22]) and for full-listing (degree-A++ in [22]) in Figure 7. The top charts present the running time of the three state-of-the-art methods for all datasets, for the mere-listing task (left) and the full-listing task (right). We can see that there is no clear winner for mere-listing: both A++ methods have a very similar duration, but core-A+- can be between 1.4 times faster and 2.4 times slower depending on the dataset. This explains why [22] and [11] did not agree on the fastest method.

On the other hand, our heuristics *Neigh*, *Check* and *Split* manage to produce orderings significantly lower C^{+-} costs. This translates directly into short running times for mere-listing with A+-. To compare our contributions with the state of the art, we take for each dataset the fastest of the three existing methods. The bottom left chart of Figure 7 shows the speedup of our methods compared to the fastest existing one.

4.4 *Split* outperforms previous full-listing methods.

For full-listing, the top right chart of Figure 7 compares the three state-of-the-art methods and shows that degree-A++ is the fastest for almost all datasets. This result is consistent with the result reported in [22], that specifically addresses full-listing. The bottom right chart shows the speedup of our three methods compared to the fastest state-of-the-art method. Note that the *Neigh* heuristic is not competitive here (speedup under one) since its ordering time is long compared to other methods.

The main result is that *Split*-A+- is always faster than previous methods. The speedup compared to existing methods is 1.16 on average, and it ranges from 1.04 on *wiki* to 1.50 on *it* dataset. *Check* also gives very good results: on medium datasets, it is a bit slower than degree-A++, but it outperforms all state-of-the-art methods on large datasets (*it, twitter, friendster, sk*), and it even beats *Split* on three of them. This hints at a transition effect: the *Check* ordering has a lower C^{+-} value but it takes $O(m)$ steps to compute, while *Split* only needs $O(n)$; for larger datasets, the listing step prevails, so the extra time spent to compute *Check* becomes profitable.

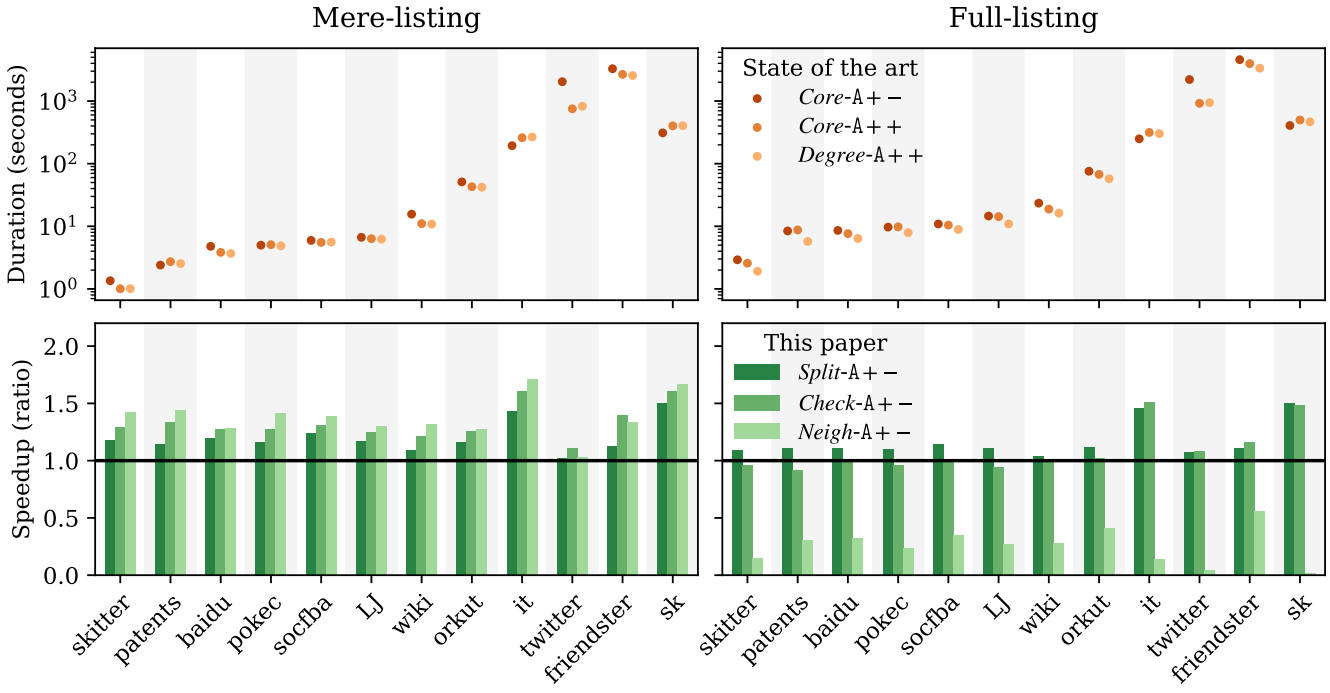


Figure 7: Comparison of state-of-the-art methods and speedup of our methods. The top charts show the runtime of the three state-of-the-art methods; depending on the dataset, the fastest method is not always the same. The bottom charts show the speedup of our three methods against the fastest existing method of each dataset. On the left, for mere-listing, we see that our three heuristics consistently outperform the three state-of-the-art methods, and that *Neigh* or *Check* are the fastest. On the right, for full-listing, *Neigh* is not efficient but *Split* is always faster than existing methods and *Check* is faster on bigger datasets.

CONCLUSION

In this work, we address the issue of in-memory triangle listing in large graphs. We formulate explicitly the computational costs of the most efficient existing algorithms, and investigate how to order vertices to minimize these costs. After proving that the optimization problems are NP-hard, we propose scalable heuristics that are specifically tailored to reduce the costs induced by the orderings. We show experimentally that these methods outperform the current state of the art for both the mere-listing and the full-listing tasks.

Our results also emphasize a limitation in the possible acceleration: while it is certainly possible to keep improving the mere-listing step, a significant part of full-listing is spent on other steps: computing the ordering, but also loading the graph or writing the output. It seems, however, that the mere-listing step takes more importance as graphs grow larger, which makes our listing methods all the more relevant for future, larger datasets. A natural extension of this work is to use similar vertex ordering heuristics in the more general case of clique listing. Formulating appropriate cost functions for clique listing algorithms is not straightforward and requires studying precisely the different possibilities to detect all the vertices of a clique.

ACKNOWLEDGMENTS

We express our heartfelt thanks to Maximilien Danisch who initiated this project. We also thank Alexis Baudin, Esteban Bautista, Katherine Byrne and Matthieu Latapy for their valuable comments. This work is funded by the ANR (French National Agency of Research) partly by the Limass project (under grant ANR-19-CE23-0010) and partly by the ANR FiT LabCom.

REFERENCES

[1] 2022. *NP-hardness of vertex ordering optimizations C^{+-} and C^{++}.* https://raw.githubusercontent.com/lecfab/volt/main/paper_proofs.pdf
[2] 2022. *Open-source c++ implementation VOLT: Vertex Orderings to List Triangles.* https://github.com/lecfab/volt
[3] Mohammad Al Hasan and Vachik S Dave. 2018. Triangle counting in large networks: a review. *WIREs Data Mining and Knowledge Discovery* (2018).
[4] Shaikh Arifuzzaman, Maleq Khan, and Madhav Marathe. 2019. Fast parallel algorithms for counting and listing triangles in big graphs. *TKDD* (2019).
[5] Vladimir Batagelj and Matjaz Zaversnik. 2003. An O(m) algorithm for cores decomposition of networks. *arXiv* (2003).
[6] Paolo Boldi and Sebastiano Vigna. 2004. The webgraph framework I: compression techniques. In *WWW*. https://law.di.unimi.it/datasets.php
[7] Raphaël Charbey and Christophe Prieur. 2019. Stars, holes, or paths across your Facebook friends: A graphlet-based characterization of many networks. *Network Science* (2019).
[8] Norishige Chiba and Takao Nishizeki. 1985. Arboricity and subgraph listing algorithms. *SIAM J. Comput.* (1985).
[9] Sarvenaz Choobdar, Pedro Ribeiro, Sylwia Bugla, and Fernando Silva. 2012. Comparison of co-authorship networks across scientific fields using motifs. In *ASONAM*. IEEE.

[10] Shumo Chu and James Cheng. 2011. Triangle listing in massive networks and its applications. In *SIGKDD*.
[11] Maximilien Danisch, Oana D. Balalau, and Mauro Sozio. 2018. Listing k-cliques in Sparse Real-World Graphs. In *WWW*.
[12] Laxman Dhulipala, Igor Kabiljo, Brian Karrer, Giuseppe Ottaviano, Sergey Pupyrev, and Alon Shalita. 2016. Compressing graphs and indexes with recursive graph bisection. In *KDD*.
[13] Lin Hu, Lei Zou, and Yu Liu. 2021. Accelerating Triangle Counting on GPU *(SIGMOD/PODS)*. https://doi.org/10.1145/3448016.3452815
[14] Alon Itai and Michael Rodeh. 1978. Finding a minimum circuit in a graph. *SIAM J. Comput.* (1978).
[15] Matthieu Latapy. 2008. Main-memory triangle computations for very large (sparse (power-law)) graphs. *Theoretical Computer Science* (2008).
[16] Eunjae Lee, Junghyun Kim, Keunhak Lim, Sam Noh, and Jiwon Seo. 2019. Preselect static caching and neighborhood ordering for BFS-like algorithms on disk-based graph engines. *USENIX*.
[17] Jure Leskovec, Lars Backstrom, Ravi Kumar, and Andrew Tomkins. 2008. Microscopic evolution of social networks. In *SIGKDD*.
[18] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data
[19] Rong-Hua Li, Sen Gao, Lu Qin, Guoren Wang, Weihua Yang, and Jeffrey Xu Yu. 2020. Ordering Heuristics for k-Clique Listing. *Proc. VLDB Endow.* (2020).
[20] Kazuhisa Makino and Takeaki Uno. 2004. New Algorithms for Enumerating All Maximal Cliques. In *SWAT*. https://doi.org/10.1007/978-3-540-27810-8_23
[21] Ron Milo, Shai Shen-Orr, Shalev Itzkovitz, Nadav Kashtan, Dmitri Chklovskii, and Uri Alon. 2002. Network motifs: simple building blocks of complex networks. *Science* (2002).
[22] Mark Ortmann and Ulrik Brandes. 2014. Triangle Listing Algorithms: Back from the Diversion. In *Proc. ALENEX*. SIAM.
[23] Nataša Pržulj. 2007. Biological network comparison using graphlet degree distribution. *Bioinformatics* (2007).
[24] Ryan A. Rossi and Nesreen K. Ahmed. 2015. The Network Data Repository with Interactive Graph Analytics and Visualization. In *AAAI*. https://networkrepository.com
[25] Mikhail Rudoy. 2017. https://cstheory.stackexchange.com/q/38274.
[26] Thomas Schank and Dorothea Wagner. 2005. Finding, counting and listing all triangles in large graphs, an experimental study. In *WEA*. Springer.
[27] Stavros Sintos and Panayiotis Tsaparas. 2014. Using strong triadic closure to characterize ties in social networks. In *SIGKDD*.
[28] Olaf Sporns, Rolf Kötter, and Karl J Friston. 2004. Motifs in brain networks. *PLoS biology* (2004).
[29] Takeaki Uno. 2012. Implementation issues of clique enumeration algorithm. *Progress in Informatics* (2012). https://doi.org/10.2201/NiiPi.2012.9.5
[30] Sergi Valverde and Ricard V Solé. 2005. Network motifs in computational graphs: A case study in software architecture. *Physical Review E* (2005).
[31] Leyuan Wang, Yangzihao Wang, Carl Yang, and John D. Owens. 2016. A Comparative Study on Exact Triangle Counting Algorithms on the GPU *(HPGP '16)*.
[32] Hao Wei, Jeffrey Xu Yu, Can Lu, and Xuemin Lin. 2016. Speedup Graph Processing by Graph Ordering *(SIGMOD)*. https://doi.org/10.1145/2882903.2915220
[33] Xiao Zhou and Takao Nishizeki. 1999. Edge-Coloring and f-Coloring for Various Classes of Graphs. *J. Graph Algorithms Appl.* (1999).