Tailored vertex ordering for faster triangle listing in large graphs

Fabrice Lécuyer* Louis Jachiet† Clémence Magnien‡ Lionel Tabourier§

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 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 38% when the ordering is already given as an input, and 16% when the ordering time is included.

1 Introduction

1.1 Context and problem statement. Small connected subgraphs are key to identifying families of real-world networks [24] and are used for descriptive or predictive purposes in various fields such as biology [32, 26], linguistics [6] or engineering [34]. In sociology in particular, characterizing networks with specific structural patterns has been a focus of interest for a long time, as it is even present in the works of early 20^{th} century sociologists such as Simmel [30]. Consequently, it is a common practice in social network analysis to describe interactions between individuals using local patterns [15, 36]. Recently, the ability to count and list small size patterns efficiently allowed the characterization of various types of social networks on a large scale [10, 8]. In particular, listing elementary motifs such as triangles and 3-motifs is a stepping stone in the analysis of the structure of networks and their dynamics [14]. For instance, the closure of a triplet of nodes to form a triangle is supposed to be a driving force of social networks evolution [20, 31].

*LIP6, Sorbonne Université, CNRS, F-75005 Paris, France
†LTCI, Télécom Paris, Institut Polytechnique de Paris, France
‡LIP6, Sorbonne Université, CNRS, F-75005 Paris, France
§LIP6, Sorbonne Université, CNRS, F-75005 Paris, France

The task of listing triangles 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 methods exist for graphs that do not fit in main memory: some use I/O-efficient accesses to the disk [11], while others partition the graph and process each part separately [4]. However, such approaches induce a costly counterpart that makes them much less efficient than in-memory listing methods. It is also worth noticing that exact or approximate methods designed for triangle *counting* [3, 35, 16] can generally not be adapted to triangle *listing*.

An efficient algorithm for triangle listing has been proposed early on in [9]. Based on the observation that real-world graphs generally have a heterogeneous degree distribution, later contributions [29, 18] showed how ordering vertices by degree or core value accelerates the listing. Such orderings create an orientation of edges so that nodes that are costly to process are not processed many times. A unifying description of this method has been proposed in [25] and it has been successfully extended to larger cliques [12, 22, 33]. 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 [7, 13] or cache optimization [37, 19]. 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 of each algorithm. We prove that finding an optimal ordering that minimizes either of these costs 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 producing orderings with low corresponding costs. Our heuristics reach a compromise between their running time and the quality of the ordering obtained, 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 [1] of all considered methods.

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 available in the full version [2]. The experiments of Section 4 show that our methods are efficient in practice and improve the state of the art.

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 [25] 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` [17], `node-iterator` [29] and `forward` [29] 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` [29], `compact-forward` [18] and `K3` [9] belong to this category, as well as some algorithms that list larger cliques [23, 12, 22].

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 [25]: a total ordering 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 3 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 3 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).

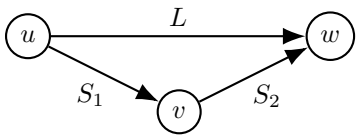

Figure 1: **Directed triangle** with the unified notations proposed in [25]. The edges are directed according to an ordering π such that $\pi_u < \pi_v < \pi_w$.

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 [29]. 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 [25, 12].

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 Algorithms 1 and 2 with the notations of Figure 1 for the vertices. 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 1 corresponds to `L+n` in [25]; we call it `A++` because of the two "+" (referring to out-degrees) involved in its complexity. The Algorithm 2 corresponds to `S`$_1$`+n` in [25]; we call it `A+-` [1].

[1] A third natural variant exists: `A--` or `S`$_2$`+n`. We ignore it here since its complexity is equivalent to the one of `A++`.

Algorithm 1 – `A++` (or `L+n`)

1: **for** each vertex v **do** $B[v] \leftarrow$ False
2: **for** each vertex w **do**
3: **for** $v \in N_w^-$ **do** $B[v] \leftarrow$ True
4: **for** $u \in N_w^-$ **do**
5: **for** $v \in N_u^+$ **do**
6: **if** $B[v]$ **then**
7: **output** triangle$\{u, v, w\}$
8: **for** $v \in N_w^-$ **do** $B[v] \leftarrow$ False

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

Algorithm 2 – `A+-` (or `S`$_1$`+n`)

1: **for** each vertex w **do** $B[w] \leftarrow$ False
2: **for** each vertex u **do**
3: **for** $w \in N_u^+$ **do** $B[w] \leftarrow$ True
4: **for** $v \in N_u^+$ **do**
5: **for** $w \in N_v^+$ **do**
6: **if** $B[w]$ **then**
7: **output** triangle$\{u, v, w\}$
8: **for** $w \in N_u^+$ **do** $B[w] \leftarrow$ False

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

Their complexities are given in Property 2.1. Since they depend on the indegree and outdegree of vertices, the choice of ordering will impact the running time of the algorithms.

PROPERTY 2.1. (COMPLEXITY OF `A++` AND `A+-`) *The time complexity of* `A++` *is* $\Theta(\sum_{u\in V} {d_u^+}^2)$. *The time 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 $n \in \mathcal{O}(m)$. 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 [25] order the vertices by non-decreasing degree or core value. In their experimental comparison, they test several algorithms as well as `A++` and `A+-`, each 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.* [12] 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 [9, 18, 23].

With these two orderings, it is possible to obtain upper-bounds for the time complexity in terms of graph properties. Chiba and Nishizeki [9] show that `K3` with degree ordering has a complexity in $\mathcal{O}(m \cdot \alpha(G))$, where $\alpha(G)$ is the arboricity of graph G. With core ordering, `node-iterator-core` [29] and `kClist` [12] have complexity $\mathcal{O}(m \cdot c(G))$, where $c(G)$ is the core value of graph G. These bounds are considered equal in [25], following the proof in [38] that $\alpha(G) \leq c(G) \leq 2\alpha(G) - 1$. However, we focus in this work on the complexities expressed in Algorithms `A++` and `A+-` 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 1 and 2. Recall that the initial graph is undirected and that the orientation of the 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 [25], and `A+-` with core ordering [12]. 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 [25], 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 3.1:

THEOREM 3.1. (NP-HARDNESS) *Given a graph $G = (V, E)$, it is NP-hard to find an ordering π on V that minimizes $C^{+-}(\pi)$ or that minimizes $C^{++}(\pi)$.*

Proof. For the hardness of C^{+-}, a proof has been proposed online [28] but never published as far as we know. In the full version of this paper [2], 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 [18, 12, 22] only take the last step into account, while [29, 25] 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 [12], mere-core-`A++` and full-degree-`A++` in [25]. We use all three methods as benchmarks in our experiments of Section 4.

Studying both tasks gives a better understanding of the phenomena at play in the speed of triangle listing. On an algorithmic point of view, mere-listing shows the impact that orderings can have on the cost of triangle listing. As the ordering time is not taken into account, a long time can be spent on finding an ordering with low cost. This also represents the situation when the dataset is then distributed to other users or used several times; for instance, in a recommendation system that requests graph motif enumeration on the fly, finding an efficient ordering once accelerates each subsequent query. On the practical side, full-listing represents the runtime for one execution: it favors quickly obtained orderings even if their induced cost is not the lowest. These differences lead to a time-quality trade-off for cost-reducing heuristics.

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

We remind that two efficient algorithms are identified in the literature for triangle listing (see Algorithms 1 and 2). Their number of operations are respectively C^{++} and C^{+-}. However, the orderings that have been considered (degree and core) induce a low C^{++} cost, but not necessarily a low C^{+-} cost.

Our goal here is therefore to design a procedure that takes a graph as input and produces an ordering π with a low induced cost $C^{+-}(\pi)$. Because of Theorem 3.1, 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 2. 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. Starting from any ordering π, the algorithm described in Algorithm 3 considers each vertex u one by one (line 3) and, for each $p \in [\![1, d_u]\!]$, it computes $C^{+-}(p)$, the value of C^{+-} when u is just after its p-th neighbor in π, as well as $C^{+-}(0)$ when u is before all its neighbors. The position p_* that induces the lowest value of C^{+-} is selected (line 5) and the ordering is updated (line 6). 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 $\mathcal{O}(d_u \log d_u)$ operations; finding the best

Algorithm 3 Neighborhood optimization (*Neigh* heuristic)

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: $p_* = \text{argmin}_{p \in [\![0, d_u]\!]} \{C^{+-}(p)\}$
6: update ordering π to put u in position p_*
7: **while** $C^{+-}(\pi) < (1 - \epsilon) \cdot C_0$

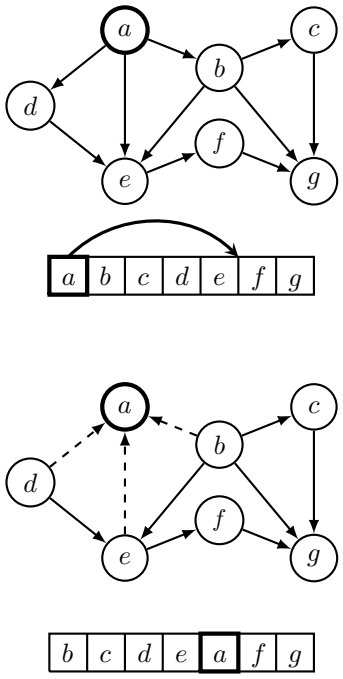

Figure 2: **Example of update in the *Neigh* heuristic**: vertex a is moved to a position among its neighbors that induces the lowest cost. The tables indicates how the ordering is updated. The edges in the DAG that are reoriented accordingly are dashed. Here, the ordering at the top has $C^{+-} = 9$ while the ordering at the bottom has $C^{+-} = 6$. For this graph, the optimal C^{+-} cost is 3 (with ordering e, g, f, a, c, d, b).

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 $\mathcal{O}(m \log \Delta)$, which leads to a total complexity $\mathcal{O}(Im \log \Delta)$ 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^{++}; it is greedy, so 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 actual task of listing triangles in practice, which is an issue for the full-listing task. We therefore propose the following faster heuristics in the case of the full-listing task.

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* whether 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 $C_e = (|N_b| + |N_?|) \cdot |N_e|$. The option with the smaller cost is selected. Sorting the vertices by degree requires $\mathcal{O}(n)$ 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 $\mathcal{O}(m + n)$, or $\mathcal{O}(m)$ assuming that $n \in \mathcal{O}(m)$.

3.3.3 *Split* heuristic. Finally, we propose a heuristic that is faster to achieve but compromises on the quality of the resulting ordering. Degree ordering has been identified as the best solution for mere-listing with algorithm-`A++` [25]. 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. With the graph of Figure 2, supposing that we start from the non-decreasing degree ordering (e, b, g, a, f, d, c), which has $C^{+-} = 7$, the *Split* method leads to (e, g, f, c, d, a, b), which has $C^{+-} = 4$. The complexity of this method is in $\mathcal{O}(n)$ 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. As this work focuses on in-memory triangle listing in large graphs, we selected networks that have between ten million and two billion edges and can therefore be loaded in the RAM of a

standard machine. These datasets are standard for evaluating graph algorithms on real-world data: most of them appear in the experiments of [25, 12] and are provided by widely-used graph collections [21, 27], while larger webgraphs come from [7]. All the tested networks are included in the experimental results below. 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 ★[21]	1,696,415	11,095,298	28,769,868
patents ■[21]	3,774,768	16,518,947	7,515,023
baidu ★[27]	2,141,301	17,014,946	25,207,196
pokec ▲[21]	1,632,804	22,301,964	32,557,458
socfba ▲[27]	3,097,166	23,667,394	55,606,428
LJ ▲[21]	4,036,538	34,681,189	177,820,130
wiki ★[21]	2,070,486	42,336,692	145,707,846
orkut ▲[21]	3,072,627	117,185,083	627,584,181
it ★[7]	41,291,318	1,027,474,947	48,374,551,054
twitter ▲[7]	41,652,230	1,202,513,046	34,824,916,864
friendster ▲[21]	124,836,180	1,806,067,135	4,173,724,142
sk ★[7]	50,636,151	1,810,063,330	84,907,041,475

4.1.2 Software and hardware.

We release a uniform open-source implementation [1] of `A++` and `A+-` algorithms, as well as the different ordering strategies that we discussed in Section 3. The code is in `c++` and compiles with `gnu make 4` and `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` with `linux suse 12.3`.

Our implementation of either triangle listing algorithm can run in parallel 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. Moreover, the goal of this work is to evaluate the impact of different methods on the speed of triangle listing, which is more straightforward to observe with a single thread.

Regarding the state of the art, the most competitive implementation available for triangle listing is `kClist` in `c` [12], which has already been shown to outperform previous programs [23, 18]. 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` in various settings and found that ours is 14% faster on average, presumably because it does not use recursion. Moreover, the paper that identified core-`A++` and degree-`A++` as the fastest methods [25] does not provide the corresponding code. Therefore, we only use our own implementation of `A+-` and `A++` in the rest of this paper: we exclusively focus on the speedup caused by the vertex ordering, separating it from the speedup originating from the implementation.

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 3, we see that the running time for a given dataset correlates almost linearly to the corresponding cost: the lines represent linear regressions. The figure only presents some of the datasets for readability; the correlation is above 0.85 on all the datasets, with a median of 0.988. Note that fluctuations in running times (due for instance to hardware constraints) cause an imperfect correlation between the executions. Ultimately, 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.

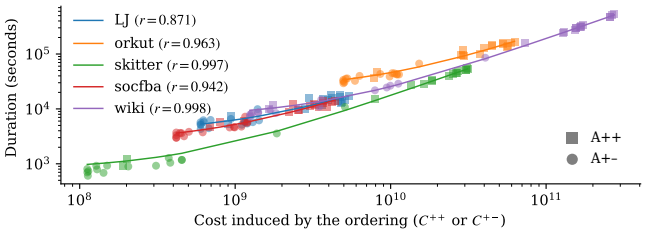

Figure 3: **Algorithm running time vs the cost induced by the ordering.** Each mark represents an ordering: circles are for cost C^{+-} and algorithm `A+-`, squares are for cost C^{++} and algorithm `A++`. Each color represents a dataset: the line of linear regressions and associated correlation coefficients r show the proportionality between cost and time.

4.3 *Neigh* outperforms previous mere-listing methods.

We compare our methods to the state of the art for mere-listing (core-`A+-` in [12] and core-`A++` in [25]) and for full-listing (degree-`A++` in [25]) in Figure 4. 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 [25] and [12] did not agree on the fastest method.

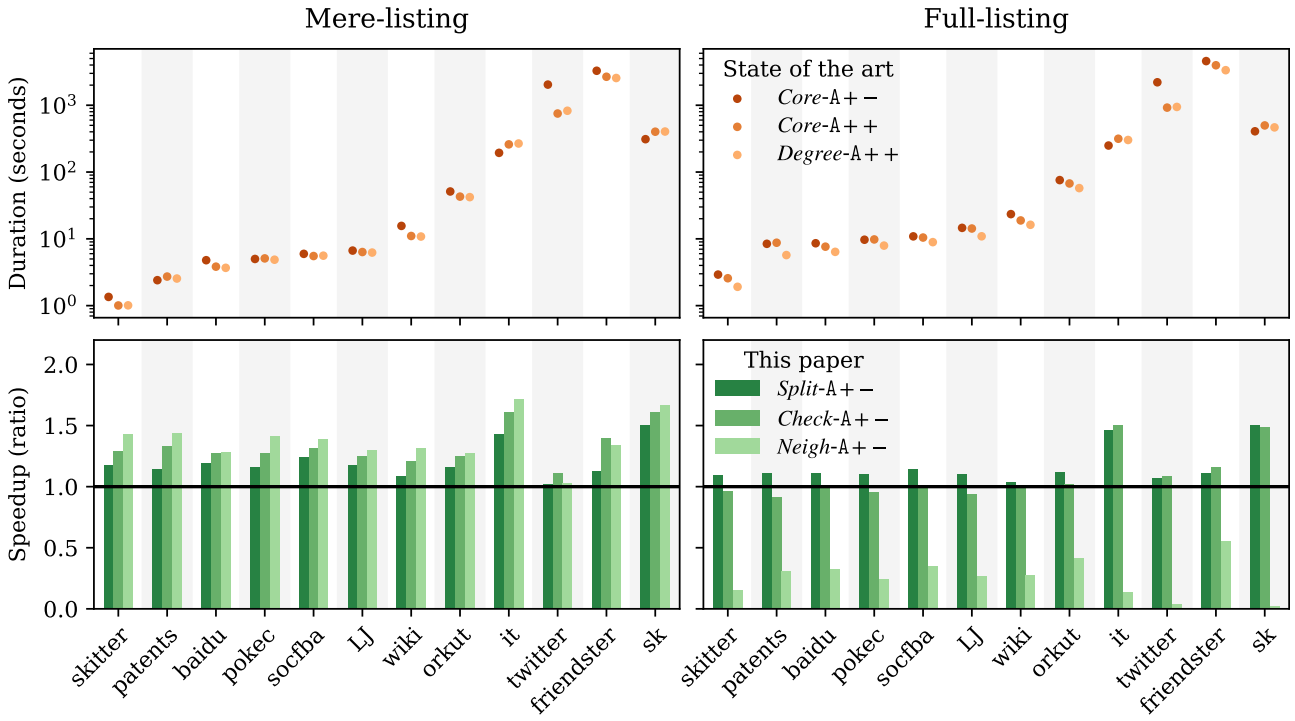

Figure 4: **Comparison of state-of-the-art methods and speedup of our methods.** The top charts show the runtime of the three state-of-the-art methods; depending on the dataset, the fastest method is not always the same. The bottom charts show the speedup of our three methods against the fastest existing 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.

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 4 shows the speedup of our methods compared to the fastest existing one. Exact runtimes of the best existing and of the methods proposed in this work are reported in Table 2.

The main result is that *Neigh*-`A+-` is always faster than the best previous method. The speedup is 1.38 on average and ranges from only 1.02 on *twitter* to 1.71 on the *it* dataset. *Check*-`A+-` is almost as good, with a 1.32 average speedup ranging from 1.10 to 1.60; it is even faster than *Neigh*-`A+-` on two of the datasets. *Split*-`A+-` is a little slower, which is expected because this ordering is designed to be obtained quickly and does not reduce C^{+-} as efficiently as our other heuristics. However it still consistently outperforms all the previous methods, with a 1.20 average speedup.

	mere-listing		**full-listing**	
dataset	existing	this paper	existing	this paper
skitter	1.00s	0.71s	1.91s	1.75s
patents	2.40s	1.67s	5.71s	5.15s
baidu	3.68s	2.87s	6.38s	5.77s
pokec	4.87s	3.44s	7.91s	7.21s
socfba	5.52s	3.98s	8.92s	7.79s
LJ	6.23s	4.79s	10.91s	9.88s
wiki	10.82s	8.22s	16.23s	15.65s
orkut	42.11s	33.09s	57.47s	51.60s
it	3m13	1m53	4m09	2m45
twitter	12m31	11m20	15m21	14m08
friendster	42m36	30m31	55m47	48m13
sk	5m10	3m06	6m47	4m31

Table 2: **Duration of triangle listing of existing methods against methods of this paper.** For each dataset, we compare the fastest state-of-the-art method against the fastest of our methods. Recall that mere-listing only takes into account the runtime of the listing algorithm (`A++` or `A+-`) while full-listing also counts the graph loading time and the ordering time.

4.4 *Split* outperforms previous full-listing methods. For full-listing, the top right chart of Figure 4 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 [25], 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 $\mathcal{O}(m)$ steps to compute, while *Split* only needs $\mathcal{O}(n)$; for larger datasets, the listing step prevails, so the extra time spent to compute *Check* becomes profitable.

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.

Preliminary investigation indicates that the mere-listing step takes more importance as graphs grow larger, which hints that our listing methods would be all the more efficient for future, larger datasets. Moreover, full-listing includes the time spent to load the graph and to compute the ordering; these two tasks can hardly be accelerated, as the complexity of *Split* is linear in the number of nodes of the graph. For this reason, we think that improving triangle listing will require to find orderings that are fast to compute and that further accelerate the mere-listing step.

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.

Acknowledgements

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] Open-source `c++` implementation, *Vertex Orderings to List Triangles*: `https://github.com/lecfab/volt`.
[2] Full version with proofs of hardness in appendix: `https://doi.org/10.48550/arXiv.2203.04774`.
[3] M. Al Hasan and V. S. Dave, *Triangle counting in large networks: a review*, WIREs Data Mining and Knowledge Discovery, (2018).
[4] S. Arifuzzaman, M. Khan, and M. Marathe, *Fast parallel algorithms for counting and listing triangles in big graphs*, TKDD, (2019).
[5] V. Batagelj and M. Zaversnik, *An O(m) algorithm for cores decomposition of networks*, arXiv, (2003).
[6] C. Biemann, L. Krumov, S. Roos, and K. Weihe, *Network motifs are a powerful tool for semantic distinction*, in Towards a Theoretical Framework for Analyzing Complex Linguistic Networks, 2016.
[7] P. Boldi and S. Vigna, *The webgraph framework I: compression techniques*, in WWW, 2004.
[8] R. Charbey and C. Prieur, *Stars, holes, or paths across your facebook friends: A graphlet-based characterization of many networks*, Network Science, (2019).
[9] N. Chiba and T. Nishizeki, *Arboricity and subgraph listing algorithms*, SIAM Journal on Computing, (1985).
[10] S. Choobdar, P. Ribeiro, S. Bugla, and F. Silva, *Comparison of co-authorship networks across scientific fields using motifs*, in ASONAM, 2012.
[11] S. Chu and J. Cheng, *Triangle listing in massive networks and its applications*, in SIGKDD, 2011.
[12] M. Danisch, O. D. Balalau, and M. Sozio, *Listing k-cliques in sparse real-world graphs*, in WWW, 2018.
[13] L. Dhulipala, I. Kabiljo, B. Karrer, G. Ottaviano, S. Pupyrev, and A. Shalita, *Compressing graphs and indexes with recursive graph bisection*, in KDD, 2016.
[14] K. Faust, *A puzzle concerning triads in social networks: Graph constraints and the triad census*, Social Networks, 32 (2010), pp. 221–233.
[15] P. W. Holland and S. Leinhardt, *Local structure in social networks*, Sociological methodology, (1976).
[16] L. Hu, L. Zou, and Y. Liu, *Accelerating triangle counting on gpu*, SIGMOD/PODS, 2021.
[17] A. Itai and M. Rodeh, *Finding a minimum circuit in a graph*, SIAM Journal on Computing, (1978).

[18] M. Latapy, *Main-memory triangle computations for very large (sparse (power-law)) graphs*, Theoretical Computer Science, (2008).
[19] E. Lee, J. Kim, K. Lim, S. Noh, and J. Seo, *Pre-select static caching and neighborhood ordering for bfs-like algorithms on disk-based graph engines*, 2019.
[20] J. Leskovec, L. Backstrom, R. Kumar, and A. Tomkins, *Microscopic evolution of social networks*, in SIGKDD, 2008.
[21] J. Leskovec and A. Krevl, *SNAP Datasets: Stanford large network dataset collection*, 2014.
[22] R.-H. Li, S. Gao, L. Qin, G. Wang, W. Yang, and J. X. Yu, *Ordering heuristics for k-clique listing*, Proc. VLDB Endow., (2020).
[23] K. Makino and T. Uno, *New algorithms for enumerating all maximal cliques*, in SWAT, 2004.
[24] R. Milo, S. Shen-Orr, S. Itzkovitz, N. Kashtan, D. Chklovskii, and U. Alon, *Network motifs: simple building blocks of complex networks*, Science, (2002).
[25] M. Ortmann and U. Brandes, *Triangle listing algorithms: Back from the diversion*, in Proc. ALENEX, SIAM, 2014.
[26] N. Pržulj, *Biological network comparison using graphlet degree distribution*, Bioinformatics, (2007).
[27] R. A. Rossi and N. K. Ahmed, *The network data repository with interactive graph analytics and visualization*, in AAAI, 2015.
[28] M. Rudoy, *https://cstheory.stackexchange.com/q/38274*, 2017.
[29] T. Schank and D. Wagner, *Finding, counting and listing all triangles in large graphs, an experimental study*, in WEA, Springer, 2005.
[30] G. Simmel, *Soziologie*, Duncker & Humblot Leipzig, 1908.
[31] S. Sintos and P. Tsaparas, *Using strong triadic closure to characterize ties in social networks*, in SIGKDD, 2014.
[32] O. Sporns, R. Kötter, and K. J. Friston, *Motifs in brain networks*, PLoS biology, (2004).
[33] T. Uno, *Implementation issues of clique enumeration algorithm*, Progress in Informatics, (2012).
[34] S. Valverde and R. V. Solé, *Network motifs in computational graphs: A case study in software architecture*, Physical Review E, (2005).
[35] L. Wang, Y. Wang, C. Yang, and J. D. Owens, *A comparative study on exact triangle counting algorithms on the gpu*, HPGP '16, 2016.
[36] S. Wasserman, K. Faust, et al., *Social network analysis: Methods and applications*, (1994).
[37] H. Wei, J. X. Yu, C. Lu, and X. Lin, *Speedup graph processing by graph ordering*, SIGMOD, 2016.
[38] X. Zhou and T. Nishizeki, *Edge-coloring and f-coloring for various classes of graphs*, J. Graph Algorithms Appl., (1999).