Tailored vertex ordering for faster triangle listing in large graphs

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

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

Louis Jachiet
Télécom Paris
Institut Polytechnique de Paris

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:

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.
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 $\pi$ 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 $\{u, v\} \in E$, $E_\pi$ 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 $N^\pi_u$ and successors $N^\pi_u$. We define the indegree $d^\pi_u = |N^\pi_u|$ and the outdegree $d^\pi_u = |N_u| - d^\pi_u$; their sum is $d^\pi_u + d^\pi_u = d_u$. 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 $e(G)$ of $G$ is the maximal $c_u$ for $u \in V$. A core ordering $\pi$ verifies $\pi_u < \pi_v \Leftrightarrow c_u \leq c_v$. Core value and core ordering can be computed in linear time [5].

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 $N^\pi_u$ 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. 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 $\Pi n$ 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 $S_1 n \in [22]$; 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 $\Theta(\sum_{u \in V} d^+_u d^-_u)$. The complexity of $A^{--}$ is $\Theta(m + \sum_{v \in V} d^+_v d^-_v)$.

Proof. In both algorithms, the boolean table $B$ requires $n$ initial values, $m$ set and $m$ reset operations, which is $\Theta(m)$ assuming that $m \in O(m)$. 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 $\Theta(d^+_u)$ operations. Similarly, in $A^{--}$, a given vertex $v$ appears in the loop of Line 4 as many times as it has a predecessor $w$; 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 $\sum_{u \in V} d^+_u d^-_u \geq \sum_{u \in V} d^+_u = m$, 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].

Note that a third natural variant exists, which would be $A^{--}$ (or $S_2 n$), but we do not consider it here since its complexity is equivalent to the one of $A^{++}$.
Tailored vertex ordering for faster triangle listing in large graphs

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 $\pi$, 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 $\pi$ are defined by:

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

$$C^{+-}(\pi) = \sum_{u \in V} d_u^+ d_u^0$$

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 $\pi$ so that their outdegree in $G_\pi$ 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 $\pi$ 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.
Algorithm Neigh
Input: graph $G$, initial ordering $\pi$, 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 $\pi$
5: $p_0 = $ position of $u$ in its neighborhood
6: $p_\pi = \operatorname{argmin}_p \{C^{++}(p)\}$
7: update ordering $\pi$ to put $u$ in position $p_\pi$
8: while $C^{++}(\pi) < (1 - \epsilon) \cdot C_0$

Figure 4: Neighborhood optimization (Neigh heuristic)

Starting from any ordering $\pi$, 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 $\pi$. The position $p_\pi$ 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 $\epsilon$ (last line). The resulting $\pi$ induces a low $C^{++}$ cost.

For a vertex $u$, sorting the neighborhood according to $\pi$ takes $O(d_u \log d_u)$ operations; finding the best position takes $\Theta(d_u)$ because it only depends on the values $d_{u,v}$ of each neighbor $v$ of $u$. With a linked list, $\pi$ is updated in constant time. If $\Delta$ 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 $O(m \log \Delta)$ if the improvement threshold $\epsilon$ 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, $\pi$ 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 $V_l$ those that are yet to place. The neighbors of $u$ are partitioned in $N_b = N_u \cap V_b$, $N_l = N_u \cap V_l$ and $N_t = N_u \cap V_e$. 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 $N_t$ as successors. In the first case, vertices in $N_l$ become successors, which induces a $C^+$ cost $C_b = |N_b| \cdot (|N_b| + |N_t|)$. In the second, the cost is $C_e = (|N_b| + |N_l|) \cdot |N_t|$. The option with the smaller cost is selected. Sorting the vertices by degree requires $O(n)$ steps with bucket sort. Maintaining the sizes of $N_b$, $N_t$, $N_l$ for each vertex requires one update for each edge. Therefore, the complexity is $O(n + n)$, 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 $\pi$. More precisely, a non-increasing degree ordering $\delta$ 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.
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 shorter 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.
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 iT LabCom.

REFERENCES

Tailored vertex ordering for faster triangle listing in large graphs

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


[16] Eunjae Lee, Junghyun Kim, Keunhak Lim, Sam Noh, and Jiwon Seo. 2019. Pre-select static caching and neighborhood ordering for BFS-like algorithms on disk-based graph engines. USENIX.


[33] Hao Wei, Jeffrey Xu Yu, Can Lu, and Xueshi Lin. 2016. Speedup Graph Processing by Graph Ordering (SIGMOD). https://doi.org/10.1145/2882903.2915220