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. (arxiv version: doi.org/10.48550/arXiv.2203.04774)

1 Introduction

1.1 Context and problem.

Small connected subgraphs are key to identifying families of real-world networks [22] and are used for descriptive or predictive purposes in various fields such as biology [24], linguistics [4], or engineering [33]. 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 20th century sociologists such as Simmel [29]. Consequently, it is a common practice in social network analysis to describe interactions between individuals using local patterns [13, 35]. Recently, the ability to count and list small size patterns efficiently allowed the characterization of various types of social networks on a large scale [8, 6]. 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 [12]. For instance, the closure of a triplet of nodes to form a triangle is supposed to be a driving force of social networks evolution [18, 30].

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 [9], while others partition the graph and process each part separately [2]. 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 [1, 5, 14] can generally not be adapted to triangle listing.

An efficient algorithm for triangle listing has been proposed early on in [7]. Based on the observation that real-world graphs generally have a heterogeneous degree distribution, later contributions [28, 16] 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 [23] and it has been successfully extended to larger cliques [10, 20, 32]. 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 [5, 11] or cache optimization [36, 17]. 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 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 in

*Corresponding authors.
†LTCl, Télécom Paris, Institut Polytechnique de Paris

*Corresponding authors.
†LTCl, Télécom Paris, Institut Polytechnique de Paris

Open-source C++ implementation available at: https://github.com/lecfab/volt
AppendixAand B The experiments of Section 3 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 $\pi$ 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 $N_u^<$ and successors $N_u^>$. We define the indegree $d_u^<$ of the set $N_u^<$ and the outdegree $d_u^>$ of $N_u^>$; their sum is $d_u = d_u^< + d_u^> = d_u$. 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 $\pi$ verifies $\pi_u \leq \pi_v \Leftrightarrow c_u \leq c_v$. Core value and core ordering can be computed in linear time [3].

2 State of the art

2.1 Triangle listing algorithms.

Ortmann and Brandes [23] 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 $\{v, (u, v)\}$ of its neighbors; if they are themselves adjacent, $\{u, v, w\}$ is a triangle. Algorithms tree-lister [15], node-iterator [28] and forward [28] 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 [28], compact-forward [14] and K3 [7] belong to this category, as well as some algorithms that list larger cliques [21] [10] [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 [23]: a total ordering $\pi$ 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 $G_\pi$ 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 $S_1$).

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 [25]. 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 [23] [10].

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 [23]: 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 [23]: 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 time complexity of $A++$ is $\Theta(\sum_{u \in V} d_u^2)$. The time complexity of $A--$ is $\Theta(m + \sum_{u \in V} d_u d^<_u)$. $^2$A third variant exists: $A--$ or $S_2+n$. We ignore it here since its complexity is equivalent to the one of $A++$. 

Figure 1: Directed triangle with the unified notations proposed in [23]. The edges are directed according to an ordering $\pi$ such that $\pi_u < \pi_v < \pi_w$.
in \(\Theta(d_v^+d_v^-)\) operations. The term \(m\) is omitted in the complexity of \(\mathcal{A}++\) as \(\sum_{u \in V} d_u^+\geq \sum_{u \in V} d_u^- = m\), but not in \(\mathcal{A}+-\) as \(\sum_{v \in V} d_v^+d_v^-\) can be lower than \(m\).

\[\square\]

### 2.2 Orderings and complexity bounds.

Ortmann and Brandes \cite{23} order the vertices by non-decreasing degree or core value. In their experimental comparison, they test several algorithms as well as \(\mathcal{A}++\) and \(\mathcal{A}+-\), each with degree ordering, core ordering, and with the original ordering of the dataset. They conclude that the fastest method is \(\mathcal{A}++\) 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. \cite{10} also use core ordering in the more general problem of listing \(k\)-cliques. For triangles \((k = 3)\), their algorithm is equivalent to \(\mathcal{A}+-\), and they show that using core ordering outperforms the methods of \cite{7,16,21}.

With these two orderings, it is possible to obtain upper-bounds for the time complexity in terms of graph properties. Chiba and Nishizeki \cite{7} show that \(K3\) with degree ordering has a complexity in \(O(m \cdot \alpha(G))\), where \(\alpha(G)\) is the arboricity of graph \(G\). With core ordering, \texttt{node-iterator-core} \cite{28} and \texttt{kClist} \cite{10} have complexity \(O(m \cdot c(G))\), where \(c(G)\) is the core value of graph \(G\). These bounds are considered equal in \cite{23}, following the proof in \cite{37} that \(\alpha(G) \leq c(G) \leq 2\alpha(G) - 1\). However, we focus in this work on the complexities expressed in Algorithms \(\mathcal{A}++\) and \(\mathcal{A}+-\) 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 \(\pi\), 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:

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

The fastest methods in the state of the art are \(\mathcal{A}++\) with core or degree ordering \cite{23}, and \(\mathcal{A}+-\) with core ordering \cite{10}. 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 \(\mathcal{A}++\). In \cite{23}, it is mentioned that core ordering performs well with \(\mathcal{A}+-\) as a side effect.

To our knowledge, no previous work has designed orderings with a low \(C^{+-}\) cost and used them with \(\mathcal{A}+-\). We will show that such orderings can lower the computational cost further. Yet, optimizing \(C^{+-}\) or \(C^{++}\) is computationally hard because of Theorem \[4\].

**Theorem 1 (NP-hardness)**

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

**Proof:** For the hardness of \(C^{+-}\), a proof was already known from \cite{20} but never published as far as we know; we give a new and simpler proof in Appendix \[A\]. We prove the result for \(C^{++}\) in Appendix \[B\].

#### 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 \cite{16,10,20} only take the last step into account, while \cite{28,23} 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 [10], mere-core-\(A^+\) and full-degree-\(A^+\) in [23]. We use all three methods as benchmarks in our experiments of Section 4.

With mere-listing, the ordering time is not taken into account, 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.

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 \(\pi\) 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 2. First, notice that changing an index \(\pi_u\) 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 \(\pi\), 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 \(\pi\), as well as \(C^+(0)\) when \(u\) is before all its neighbors. The position \(p_u\) 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 \(\epsilon\) (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 \(d^+_v\) and \(d^-_v\) 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 \(O(Im \log \Delta)\) 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 \(C^+\); 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

![Figure 2: Example of update in the Neigh heuristic](image)

Algorithm 3 Neighborhood optimization (Neigh heuristic)

**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: \(p_u = \arg\min_{p \in [0,d_u]} \{C^+(p)\}\)
6: update ordering \(\pi\) to put \(u\) in position \(p_u\)
7: while \(C^+(\pi) < (1 - \epsilon) \cdot C_0\)
Section [3] 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 [3]. 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_2 those that are yet to place. The neighbors of u are partitioned in N_b = N_u ∩ V_b, N_e = N_u ∩ V_e and N_2 = N_u ∩ V_2. We consider two options to place u: either just after the vertices in V_b (π_u = |V_b| + 1), or just before the vertices in V_e (π_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_2 become successors, which induces a C^+ cost C_b = |N_b| · (|N_e| + |N_2|). In the second, the cost is C_e = |N_b| · |N_2| · |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 \(N_b\), \(N_e\), \(N_2\) 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 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++ [23]. 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. 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 \(O(n)\) 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 | ▲ | 1,696,415 | 11,095,298 | 28,769,868 |
| baidu | ▲ | 3,774,768 | 16,518,947 | 7,515,023 |
| p0licing | ▲ | 2,141,301 | 17,014,946 | 25,207,196 |
| socfb | ▲ | 1,632,804 | 22,301,964 | 32,557,458 |
| socfb | ▲ | 3,097,166 | 23,667,394 | 55,606,428 |
| socfb | ▲ | 4,036,538 | 34,681,189 | 177,820,130 |
| wiki | ▲ | 2,070,486 | 42,336,692 | 145,707,846 |
| orkut | ▲ | 3,072,627 | 117,185,083 | 627,584,181 |
| twitter | ▲ | 41,291,319 | 1,027,474,947 | 48,374,551,054 |
| friendster | ▲ | 124,836,180 | 1,806,067,135 | 4,173,724,142 |
| sk | ▲ | 50,636,151 | 1,810,063,330 | 84,907,641,475 |

4.1.2 Software and hardware.

We release a uniform open-source implementation of A++ and A++ algorithms, as well as the different ordering strategies that we discussed in Section 3. 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 intex e5-4650L 2.6 GHz, 128Gb ram running linux suse 12.3.

Regarding the state of the art, the most competitive implementation available for triangle listing is kClist in c [11], which has already been shown to outperform previous programs [21, 15]. 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 [23] 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

https://github.com/lecfab/volt
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.

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 [10] and core-$A^{++}$ in [23]) and for full-listing (degree-$A^{++}$ in [23]) 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 [23] and [10] 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 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 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.

| dataset | mere-listing existing | this paper | full-listing 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 |
| p2p    | 4.87s | 3.44s | 7.91s | 7.21s |
| socfb   | 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 | 33.08s | 57.47s | 51.68s |
| 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 [23], 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.
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.

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.

**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 LIMASS project (under grant ANR-19-CE23-0010) and partly by ANR FiT LabCom.

**References**

[1] M. Al Hasan and V. S. Dave. Triangle counting in large networks: a review. *WIREs Data Mining and Knowledge Discovery*, 2018.

[2] S. Arifuzzaman, M. Khan, and M. Marathe. Fast parallel algorithms for counting and listing triangles in big graphs. *TKDD*, 2019.

[3] V. Batagelj and M. Zaversnik. An O(m) algorithm for cores decomposition of networks. *arXiv*, 2003.

[4] 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.

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

[6] R. Charbey and C. Prieur. Stars, holes, or paths across your facebook friends: A graphlet-based characterization of many networks. *Network Science*, 2019.

[7] N. Chiba and T. Nishizeki. Arboricity and subgraph listing algorithms. *SIAM Journal on Computing*, 1985.

[8] S. Choobdar, P. Ribeiro, S. Bugla, and F. Silva. Comparison of co-authorship networks across scientific fields using motifs. In *ASONAM*, 2012.

[9] S. Chu and J. Cheng. Triangle listing in massive networks and its applications. In *SIGKDD*, 2011.
A NP-hardness of the $C^{+ -}$ problem

Given a graph $G$ and an order $\prec$ on the vertices of $G$, we define $\text{succ}_\prec(u)$ (respectively $\text{pred}_\prec(u)$) as the set of neighbors $v$ of $u$ such that $u \prec v$ (resp. $v \prec u$). For any subset of vertices $W$, we note $C^{+ -}_\prec(W) = \sum_{u \in W} |\text{succ}_\prec(u)| \cdot |\text{pred}_\prec(u)|$. Using this definition we formalize the following problem:

**Problem 1 ($C^{+ -}$)** Given an undirected graph $G = (V, E)$ and an integer $K$, is there an order $\prec$ on the vertices such that $C^{+ -}_\prec(V) \leq K$?

**Problem 2 (NAE3SAT+) Not-All-Equal Positive Three-Satisfiability.** Given a formula $\phi = c_1 \land \cdots \land c_m$ in conjunctive normal form where each clause consists in three positive literals, is there an assignment to the variables satisfying $\phi$ such that in no clause all three literals have the same truth value?

The NAE3SAT+ problem is known to be NP-complete by Schaefer’s dichotomy theorem [27]. We will show that this problem can be reduced to the $C^{+ -}$ problem, thus proving that $C^{+ -}$ is NP-hard. Note that a proof was given in-hard [26] but, as far as we know, it has never been published. We give a new simpler proof of the following theorem:
Theorem 2 \( C^{++} \) is NP-hard.

Definition 2 Let \( \phi \) be an instance of \( \text{NAE3SAT}^+ \) with variables \( x_1, \ldots, x_n \) and clauses \( c_1, \ldots, c_m \), where clause \( c_j \) is of the form \( l_1^j \lor l_2^j \lor l_3^j \). We define a graph \( G_\phi \) by creating three connected vertices \( L_1^j, L_2^j, L_3^j \) representing the literals of each clause \( c_j \); additionally, a vertex \( X_i \) is created for each variable \( x_i \) and connected to all the \( L_i^j \) such that \( l_i^j = x_i \). More formally, \( G_\phi = (V_\phi, E_\phi) \) with:

- \( V_\phi = \{X_i \mid i \in [1,n]\} \cup \{L_1^j, L_2^j, L_3^j \mid j \in [1,m]\} \)
- \( E_\phi = \\{(L_1^j, L_2^j), (L_1^j, L_3^j), (L_2^j, L_3^j) \mid j \in [1,m]\} \cup \\{\{X_i, L_i^j\} \mid x_i = l_i^j\} \)

... \( X_{i_1} - \underbrace{L_1^j}_{\text{True variables}} \)

... \( X_{i_2} - L_2^j \)

Proposition 1 (\( \implies \)) Given an instance \( \phi \) of \( \text{NAE3SAT}^+ \) with \( m \) clauses and the associated graph \( G_\phi \), if \( \phi \) is satisfiable then there exists an order \( \prec \) on \( V_\phi \) such that \( C^{++}_\prec (V_\phi) \leq 2m \).

Proof: Let \( \phi \) be a satisfiable instance of \( \text{NAE3SAT}^+ \) with the above notations. Take a valid assignment and let us note \( k \) the number of variables set to true. There exist indices \( i_1, \ldots, i_k \) such that \( x_{i_1}, \ldots, x_{i_k} = \text{true} \) and \( x_{i_{k+1}}, \ldots, x_n = \text{false} \), and for each clause \( c_j \) there are indices \( t_j, a_j, f_j \in \{1,2,3\} \) such that \( l_t^j = \text{true} \), \( l_a^j = \text{false} \), and \( l_f^j \) has any value. Now construct the following order on \( V_\phi \), so that true variables come first, then in each clause the false literal comes before the true one, and the false variables are at the end:

\[
X_1 \prec \cdots \prec X_k \quad \text{True variables}
\]

\[
\prec L_1^j \prec \cdots \prec L_m^j \quad \text{False literals}
\]

\[
\prec L_1^j \prec \cdots \prec L_m^j \quad \text{Other literals}
\]

\[
\prec L_1^j \prec \cdots \prec L_m^j \quad \text{True literals}
\]

\[
\prec X_{k+1} \prec \cdots \prec X_n \quad \text{False variables}
\]

The weighted-\( C^{++} \) problem. For the sake of simplicity our proof of completeness will rely on a second novel problem, the weighted-\( C^{++} \) problem, and we will show that \( C^{++} \) is NP-complete by exhibiting first a reduction between \( C^{++} \) and weighted-\( C^{++} \) and then a second reduction between weighted-\( C^{++} \) and the Set Cover problem (a well-known NP-complete problem). We now present the weighted-\( C^{++} \) problem:

Problem 4 (weighted-\( C^{++} \)) Given an undirected graph \( G = (V,E) \), a vertex-weighting function \( w \colon V \to \mathbb{N} \) and an integer \( K \), does there exist an order \( \prec \) of the vertices such that \( \sum_{u \in V} |\text{succ}_\prec (u)| + w(u) \leq K \)?
Terminology. Given a graph $G$ with the vertex weighting function $w$ and an order $\prec$, the cost is the function $\sum_{u \in V} (|\text{succ}_\prec(u)| + w(u))^2$ applied to the graph with that order. The optimal cost of a graph is the minimal cost achievable by any order. Notice that an instance of the weightless problem can be viewed as an instance of the weighted problem where all weights are 0.

B.1 Optimality criteria for orders.

One difficulty of the reduction proofs is to show that an order necessarily behaves in a controlled way. We see in this section several criteria that ensures that some order has an optimal cost.

We define the notion of multiset of costs that will help expressing optimality criteria for orders. Given a graph $G$ and an order $\prec$, the multiset of costs $MC(G, \prec)$ is the multiset of cost of the linear cost of the quotient of the Euclidean division of the linear cost and $b$ times $d$.

Proof: $\prec$ (weighted or not) is not really useful for weightless cost but a lower cost than $MC$ optimal. □

Property 3 When there exists some $d \in \mathbb{N}$ such that $MC(G, \prec)$ contains only the values $d$ and $d + 1$ then the order $\prec$ is optimal.

Proof: Let us consider an order $\prec$ as described above: it only contains $a$ times $d$ and $b$ times $d + 1$ for some $d$, and let us consider any optimal order $\prec'$. Suppose that the multiset $MC(G, \prec')$ contains $e$ and $f$ such that $e < f - 1$. Then replacing them by $e + 1$ and $f - 1$ reduces the cost because $(e^2 + f^2) - ((e + 1)^2 + (f - 1)^2) = 2(f - 1 - e) > 0$. By iteratively applying this operation we end up with a multiset $M$ that has the same size and the same linear cost but a lower cost than $MC(G, \prec')$ and contains $a'$ times $d'$ and $b'$ times $d' + 1$ for some $d'$.

Without loss of generality we can suppose that there is at least one $d$ in $MC(G, \prec)$ which means that $d$ is the quotient of the Euclidean division of the linear cost by $|V|$. For the same reason, $d'$ is the quotient of the Euclidean division of the linear cost of $MC(G, \prec')$ by $|V|$. Because the linear cost of $MC(G, \prec)$ does not depend on $\prec$ this proves that $d' = d$ as well as $a = a'$ and $b = b'$, which in turn implies that the costs of $MC(G, \prec)$ and $MC(G, \prec')$ are similar, and thus $\prec$ is optimal. □

While the property above is true for any graph (weighted or not) it is not really useful for weightless graphs because, in a weightless graph, the last vertex $u$ that we eliminate in the order $\prec$ has $|\text{succ}_\prec(u)| = 0$, and more generally the vertex $u$, which is ordered in the $i$-th position from the end, has $|\text{succ}_\prec(u)| < i$. The following property handles this case:

Property 4 For a weightless graph, when there exists $d \in \mathbb{N}$ such that $MC(G, \prec)$ contains all integers from $0$ to $d + 1$ and at most once the integers $0$ to $d - 1$, then $\prec$ is an optimal order.

Proof: The proof of optimality is similar to the proof of property 3 when the property does not hold, we can find two elements $v_i$ and $v_j$ with $v_i + 2 \leq v_j$ and we can diminish the cost by setting $v_i = v_i + 1$ and $v_j = v_j - 1$. □

Finally, let us introduce the notion of marginal cost for a multiset.

Definition 3 (Marginal cost) We introduce the marginal cost to measure how much a multiset deviates from the optimal repartition (as given in property 3). Formally, given a multiset $M$ of size $n$, we can compute $d$ such that the linear cost of $M$ is $n \times d + v$ where $1 \leq v \leq n$. The marginal cost $c_m$ of $M$ is then:

$$c_m = \sum_{u \in M} \max(0, u - (d + 1))$$

Note that we can equivalently define the marginal cost of $M$ as $\sum_{u \in M} u - \sum_{u \in M} \min(d + 1, u)$. We know from property 3 that the multiset $M'$ that minimizes the cost with the same linear cost and the same size only contains $d$ and $d + 1$ (with at least one $d + 1$ since $v > 0$). In other words the marginal cost counts the number of elements larger than needed and how much they go over the average cost: if we have a $d + 2$ it counts for 1, if we have a $d + 5$ it counts for 4, etc.

Note that the marginal cost cannot be used directly to decide if an order is optimal. Indeed, consider the two following multisets: $M$ composed of nine times the value 10 and one time the value 11 and $M'$ composed of nine times the values 11 and one time the value 2. They have the same size, the same linear cost and the same marginal cost (which is 0) but $M$ has a lower squared cost than $M'$.

The following property describes the minimal cost among all the multisets with the same size, the same linear cost and the same marginal cost:

Property 5 Among all the multisets that have a size $n$, a linear cost of $d \times n + v$ with $1 \leq v \leq n$ and a marginal cost of at least $k$ (with $2k < v$), then the ones achieving the minimal cost are composed of $k$ times the value $d + 2$, $v - 2k$ times the value $d + 1$ and $(n - v + k)$ times the value $d$.

Proof: Take such a multiset $M$ with size $n$, linear cost of $d \times n + v$ and marginal cost of at least $k$, satisfying $2k < v$. Suppose that $M$ contains a value $d - i$ with $i > 0$. Because the linear cost of $M$ is strictly larger than $d \times n$ we can find at least one value $d + j$
with \( j > 0 \) such that diminishing \( d + j \) to \( d + j - 1 \) keeps the marginal cost above \( k \).

Indeed, in a first case, there is at least one value \( d + 1 \) in \( M \), and diminishing this value to \( d \) does not affect the marginal cost \( c_m \). In the other case, let us consider the sum \( \sum_{u \in M} \min(d + 1, u) \), by definition of the marginal cost, there are no more than \( c_m \) elements in \( M \) larger than \( d + 1 \). All other elements are at most \( d \) with at least one \( d - i < d \). So, we have \( \sum_{u \in M} \min(d + 1, u) < nd + c_m \) and using \( c_m = \sum_{u \in M} u - \sum_{u \in M} \min(d + 1, u) \), we find that

\[
c_m > nd + v - (nd + c_m) = 2c_m > v
\]

and as we have made the assumption that \( v > 2k \), we have \( c_m > k \). Consequently, we can also find in this case a value \( d + j \) with \( j > 0 \) such that diminishing \( d + j \) to \( d + j - 1 \) keeps the marginal cost above \( k \).

We can deduce from this observation that the multi-set of size \( n \), with a linear cost \( d \times n + v \) with \( 1 \leq v \leq n \) and a marginal cost of at least \( k \) (with \( 2k < v \)) that achieves the minimal cost has \( k \) times the value \( d + 2 \), \( v - 2k \) times the value \( d + 1 \) and \((n - v + k)\) times the value \( d \).

This property can then be used to compare multisets and is summarized by the following property:

**Property 6** When \( M \) has a marginal cost of \( k \) then the cost of \( M \) is at least \( 2k \) larger than the balanced distribution (as given by property [4]). This \( 2k \) bound is reached for the optimality criterion described in property [4].

**Proof:** As seen before the optimal can be reached by taking two values \( i, j \in M \) with \( i + 2 \leq j \) and changing them to \( i + 1 \) and \( j - 1 \). This balancing operation can reduce by at most 1 the marginal cost but reduces the cost by \( i^2 + j^2 - (i + 1)^2 - (j - 1)^2 = 2(j - i) - 2 \) and since \( j - i \geq 2 \) this means the reduction is at least 2 and exactly 2 when \( i + 2 = j \). Since we need at least \( k \) balancing operations to reach the optimal, this gives us at least a \( 2k \) reduction of the cost to reach the optimal. Notice that when dealing with an optimal multi-set in the sense of property [4] we only combine a \( d \) with a \( d + 2 \) which gives us the exact bound. Conversely if we are not in the case of [5] we will have to combine something below (or equal to) \( d \) with something larger than \( d + 3 \) or do a combination that does not diminish the marginal cost (such as combining \( d + 1 \) and \( d + 3 \)).

\[\Box\]

### B.2 Reduction between weighted-\( C_{++} \) and \( C^{++} \).

Any instance of \( C^{++} \) can be seen as an instance of weighted-\( C_{++} \) where the weights are set to 0. For that purpose, the idea is to take a vertex \( u \) with some non null weight \( w(u) \), link \( u \) to \( w(u) \) vertices \( v_1, \ldots, v_{w(u)} \) and make sure that we can guarantee that \( u \) appears before all the \( v_1, \ldots, v_{w(u)} \) in any optimal order. We will thus exhibit a family of graphs to create such \( v_i \) vertices before showing that these \( v_i \) vertices can always appear after \( u \) in the order. Finally we will prove the full reduction.

#### B.2.1 The \( L_d \) family of graphs.

Let us consider the graph \( L_d \) parameterized by \( d \in \mathbb{N} \) that contains a \((d + 1)\)-clique \( K^d \) composed of the vertices \( K^d_0, \ldots, K^d_d \). One vertex \( e_d \) that has \( d \) neighbors \( v^d_0 \) \( \ldots \) \( v^d_d \) and such that there is an edge between each \( v^d_i \) and each vertex of \( K^d \). Consequently, there are three types of vertices in \( L_d \): the vertex \( e_d \), the vertices of type \( V \) (the \((v^d_i)_i\)) and the vertices of type \( K \) (the \((K^d_i)_i\)). Here is a depiction of \( L_d \):

![Diagram of the \( L_d \) family of graphs]

**Best cost \( C_d \) for \( L_d \).** In the weightless case, the best cost \( C_d \) for \( L_d \) is induced by the order that starts with \( e_d \) followed by the \( v^d_i \) nodes and finally by the \( K^d_i \) nodes. Indeed, in that case the cost is \( d^2 \) for \( e_d \), \((d + 1)^2 \) for each \( v^d_i \) and \( i^2 \) for \( K^d_i \) (supposing we start with \( K^d_d \) and end with \( K^d_0 \)). This is optimal by virtue of property [4].

**Best cost for \( L_d \) with a weight 1 on \( e_d \).** If we add a weight 1 on \( e_d \) then the best cost can be achieved with the same order but this time the cost of \( e_d \) is increased from \( d^2 \) to \((d + 1)^2 \) which means an increase of \( 2d + 1 \). In other words, in that case, the best cost is \( C_d + 2d + 1 \). Note that here, the optimality cannot be deduced directly from property [4] as the property only applies to weightless graphs. However we prove that there is an optimal order starting with \( e_d \).

For that, consider any order \( \prec \) and let us show that \( \prec \) can always be improved to an order that places the vertex \( e_d \) in first position.

The order \( \prec \) ranks three types of vertices: \( e_d \), \( V \) nodes and \( K \) nodes, according to the description above. Let us first suppose that there is a vertex of type \( K \) before a vertex of type \( V \) before the vertex \( e_d \). In that case the first \( i \) vertices are of type \( V \) (we can have \( i = 0 \)), then we have \( j + 1 \) vertices of type \( K \) and then one vertex of type \( V \). Let us consider how the cost changes by exchanging this last \( K \) with this last \( V \), i.e., to change from \( V^iK^{j+1}KV \) to \( V^{i+1}K^{j}VK \). It is clear that the cost changes only for the exchanged \( V \) and \( K \). Before the exchange the cost of \( V \) was \((d - j)^2 \) and after it is \((d - j + 1)^2 \) whereas for \( K \) it was \((2d - i - j)^2 \).
and after it is \((2d - i - j - 1)^2\). Overall, if \(\Delta C\) is the difference between the cost before and the cost after the exchange, we have:

\[
\Delta C = (2d - i - j)^2 - (2d - i - j - 1)^2 \\
+ (d - j)^2 - (d - j + 1)^2 \\
= 2d - 2i - 4 \\
= 2(d - i - 2)
\]

Therefore, unless \(i + 1 = d\), the cost decreases which means that we can always move the vertices at the beginning except for maybe one to improve the cost of the order. In the end we have that the beginning of an optimal sequence can be restricted to the form \(V_1^lK^1e_1d\) or \(V^d-1K^lVe_d\). In the first case, transforming \(V_1^lK^1e_1d\) into \(e_dV^lK^1\) decreases the score by \((i^2 + i)\). In the second case, transforming \(V^d-1K^lVe_d\) into \(e_dV^{d-1}K^lV\) decreases the score by \(d^2 + d - 2l\) (which is \(\geq 0\) because \(l \leq d\)). Thus, we can move in all cases \(e_d\) at the beginning of the order to improve the related cost.

We have proved that the best cost can be achieved by placing \(e_d\) at the beginning of the order. As the best cost \(C_d\) for the rest of the order is unaffected by the cost of the elimination of \(e_d\) first, we have that the best cost is \(C_d + 2d + 1\).

B.2.2 Partitioned graphs.

Let us consider a graph \(G\) composed of two subgraphs \(G_1\) and \(G_2\) plus exactly one edge \(\{e_1, e_2\}\) with \(e_1 \in V_1\) and \(e_2 \in V_2\). Any order \(\prec\) on \(V\) induces an order on \(V_1\), an order on \(V_2\) and an order between \(e_1\) and \(e_2\). If another order \(\prec'\) induces the same order on \(V_1\), the same order on \(V_2\) and the same order between \(e_1\) and \(e_2\), it has the same cost as \(\prec\). Therefore, an optimal order for \(G\) can be seen as either an optimal for \(G_1\) and an optimal order for \(G_2\) where we add a weight of 1 on \(e_2\) (if \(e_2\) precedes \(e_1\)), or an optimal order for \(G_2\) and an optimal order for \(G_1\) where we add a weight of 1 on \(e_1\) (if \(e_1\) precedes \(e_2\)).

As a result, we obtain the following property:

**Property 7** If adding a weight 1 on \(e_1\) in \(G_1\) increases the best cost of \(G_1\) by \(x\) and if adding a weight 1 on \(e_2\) increases the best cost of \(G_2\) by at most \(x\), then the best cost of \(G\) is equal to the best cost of \(G_1\) plus the best of \(G_2\) where we add a weight of 1 on \(e_2\).

B.2.3 Finishing the reduction.

**Property 8** Let \((G, K)\) be an instance of the weighted problem, we can compute an equivalent instance of the weightless problem in a time polynomial in the number of edges and vertices in \(G\) plus the sum of weights in \(G\).

**Proof:** If all the weights in \(G\) are zeros, the result is immediate. Let us suppose that there is a vertex \(u\) with a weight \(w(u) > 0\) and a degree \(d - w(u)\). Let us consider the graph \(G'\) composed of \(G\) but where the weight of \(u\) is reduced by 1 plus a fresh copy of \(L_d\) and an edge between \(u\) and \(e_d\). We claim that the best cost of \(G'\) is lower than \(K + C_d\) if and only if the best cost of \(G\) is lower than \(K\).

Indeed, we have shown that the graph \(L_d\) is such that adding a weight 1 on \(e_d\) increases the best cost from \(C_d\) to \(C_d + 2d + 1\). We also know that the sum of the degree of \(u\) plus its cost is \(d\) therefore for any order \(\prec\) adding a weight 1 on \(u\) increases the cost of \(\prec\) by at most \(2d + 1\). By applying property \(\square\) where \(G\) has the role of \(G_2\) (\(u = e_2\)) and \(L_d\) of \(G_1\) \((e_d = e_1)\) and \(x = 2d + 1\), we obtain that the best cost of \(G'\) is the best cost of \(G\) where node \(u\) has a weight increased by 1. In other words, it is equivalent in terms of best cost to handle the graph \(G\) or to handle the graph \(G'\) where the weight of node \(u\) has been decreased by 1 unit.

By applying \(\sum_u w(u)\) times this property we obtain an instance \((G', K')\) of the weightless problem which is equivalent to the instance of the weighted problem \((G, K)\). This new instance has \(\sum_u w(u) \times |L_{\deg(u)} + w(u)|\) more vertices than the original one, but this is still polynomial in the size of \(G\) plus the sum of weights and the resulting instance can be computed in polynomial time.

This proves that if the weighted-\(C^+\) problem is strongly NP-hard, the \(C^{++}\) problem is also NP-hard.

B.3 Reduction between the weighted-\(C^{++}\) and Set Cover.

Our reduction for the weighted case will be a strong reduction, meaning the version of the problem where the weights are polynomial in the size of the graph is still NP-hard. It will be based on the Set cover problem. We recall here the definition of this problem and invite the reader to check the literature for a proof of its NP-completeness:

**Problem 5 (Set cover)** Given two integers \(n, k\), we denote \(U\) the set of elements \(\{1, \ldots, n\}\). Let \(P\) be a set of sets of elements of \(U\), does there exist a subset \(P' \subseteq P\) of size \(k\) such that \(\cup_{S \in P'} S = U\)?

Let us fix an instance \((P, n, k)\) of the Set Cover problem asking whether we can find \(k\) sets \(S_1, \ldots, S_k\) in \(P\) such that \(S_1 \cup \cdots \cup S_k = U\). We suppose, without loss of generality, that the instance is not trivial in the sense that \(|P| \geq k\) (there are at least \(k\) sets in \(P\)), \(\cup_{S \in P} S = U\) (each integer in \(U\) is contained in at least one \(S \in P\)) and that all sets \(S \in P\) are such that \(S \subseteq U\).

Let us exhibit a weighted graph \(G\) and a value \(V\) such that best cost for \(G\) is less than \(V\) if and only if \(\{1, \ldots, n\}\) can be covered with \(k\) sets from \(P\).

B.3.1 Construction of a weighted-\(C^{++}\) instance from a Set Cover instance.

Our reduction will provide a graph \(G\) with a weight function \(w\) depending on a parameter \(d\) such that the Set Cover instance has a solution if and only if the best order has a multiset of costs containing at most \(k\)
values $d+2$ and all other values are either $d$ or $d+1$ (we will explicit later the values of $V$ and $d$).

**Vertices of $G$.** In $G$ the vertices are: a special vertex $A$, $n$ vertices $e_1, \ldots, e_n$ one for each $i \in \{1, \ldots, n\}$, $\ell$ vertices, $s_1, \ldots, s_{\ell}$ with one vertex $s_j$ for each set $S_j \in P$ and finally three vertices $a_j^i, b_j^i, c_j^i$ for each $i \in S_j$.

**Edges of $G$.** The vertex $A$ has an edge with all vertices of the form $s_j$ or $e_i$. For a pair $(i, j)$ with $i \in S_j$, both $a_j^i$ and $b_j^i$ have an edge with $s_j$ and $c_j^i$; in turn $c_j^i$ has an edge with $e_i$.

Overall the graph $G$ looks like this:

![Graph Diagram]

**Weights of vertices in $G$.** Recall that the cost of a vertex is the sum of its degree and its weight. In $G$, we set the weights so that each vertex has a cost of $d+2$, except for the $c_j^i$ which have a cost of $d+3$ and the vertex $A$ which has a cost of $d+1+n+k$. Parameter $d$ needs to be large enough so that all weights are positive. This is not constraining for vertices $a_j^i, b_j^i$ and $c_j^i$. Vertices $s_j$ have degree 1 plus twice the number of $i$ appearing in set $S_j$, i.e., $1 + 2 \times |S_j|$, so it suffices that $d > 2 \times |S_j|$ for all $i \in S_j \in P$. Vertices $e_i$ have degree 1 plus the number of $S_j$ sets where $i$ can be found, which is at most $1+\ell$. Vertex $A$ has degree $\ell+n$. Having the additional condition $d > \ell$ is sufficient to guarantee the constraint on $A$ and $e_i$ vertices.

**Value of $V$.** As we will show, when there is a Set Cover with $k$ sets then we have an order $<$ for $G$ such that $MC(G, \prec)$ contains $k$ times the value $d+2$ (corresponding to the $k$ selected sets), $\sum_{S \in P} |S| - n$ times the value $d$ and all the other values are $d+1$. It implies that the cost $V = k(d+2)^2 + (\sum_{S \in P} |S| - n)d^2 + r(d+1)^2$, where $r$ is the number of vertices in $G$ minus $k$ and minus $(\sum_{S \in P} |S| - n)$.

Note that, per property [3] this value $V$ corresponds to the minimal cost for an order that has a marginal cost of $k$. Conversely, we will show that if there is a solution with a marginal cost of $k$ or less then there is a Set Cover with $k$ sets, proving that it is a reduction.

Note that this converse direction is stronger than what is needed as there exists multisets with a marginal cost of $k$ that do not match the minimal cost.

The general intuition underlying the equivalence between a solution (if any) of the Set Cover problem and a solution of the corresponding weighted-$C^{++}$ problem is the following. The first $k$ vertices $s_j$ selected in the elimination order correspond to the $S_j$ sets that cover $U$. Indeed, each of these vertices generate exactly a marginal cost of 1 and all other nodes according to the elimination order will not generate any marginal cost if we can eliminate all $e_i$ nodes without adding any marginal cost. This condition is met if deleting the first $s_j$ nodes allows to decrease the cost of all $e_i$ nodes by (at least) 1 unit, which means that we have deleted at least one triplet $a_j^i, b_j^i, c_j^i$ related to node $e_i$. If so, we have found an elimination order with cost $V$ as well as $k$ sets $S_1 \ldots S_k \in P$ which cover $U$.

**B.3.2 Proof that a solution to Set Cover implies a solution to $C^{++}$.**

Suppose that we have a solution to Set Cover with the sets $S_{j_1}, \ldots, S_{j_k}$. Let us prove that our graph $G$ has an elimination order where the cost of each vertex is $d$ or $d+1$ or $d+2$ but with only $k$ vertices with cost $d+2$.

The elimination order can be built by having $j$ going through $j_1, \ldots, j_k$. For each $j$ value, we eliminate first $s_j$ for a cost of $d+2$, then we go through $i \in S_j$ and eliminate the corresponding $a_j^i$ and $b_j^i$ vertices (both at cost $d+1$ once $s_j$ has been removed). Then we eliminate $c_j^i$ (for a cost of $d$ if $e_i$ is already eliminated and $d+1$ otherwise). Finally, if $e_i$ has not yet been eliminated by a previous $j$ value, we eliminate it for a cost of $d+1$.

Once we have done this, the vertex $A$ has lost $k+n$ neighbors: all the $e_i$ and the $k$ vertices $s_j$ that we have selected. Its remaining cost is $d+1$ so we eliminate it, which in turn means that all the remaining $s_j$ have a cost of $d+1$ and we can eliminate them all (with their $a_j^i, b_j^i$ and $c_j^i$ attached).

Overall the cost of this elimination order is exactly $V$.

**B.3.3 Proof that a solution to weighted-$C^{++}$ implies a solution to Set Cover.**

Suppose that we have an order $\prec$ such that the total cost is below $V$. Since $V$ is the optimal cost for a marginal cost of $k$, the order $\prec$ cannot have a marginal cost higher than $k$ otherwise its cost would be higher than $V$ (see property [3]). Knowing that $\prec$ has a marginal cost of at most $k$, we will extract a solution to the corresponding Set Cover instance.

First we notice that when $A$ is eliminated, its cost is $d+1+k-E_s+E_n+R_n$ where $E_s$ the number of $s_j$ eliminated, and $R_n$ is the number of $e_i$ remaining. However, as long as $A$ is not eliminated, $E_s$ is less than (or equal to) the marginal cost of all the vertices eliminated before $A$. Indeed, if $s_j$ is eliminated while $A$ is still present it is because we have paid a marginal
cost at least 1 to eliminate directly one of $s_j$ or $a^i_j$ or $b^i_j$ or $c^i_j$ for $i \in S_j$. That is true because if $A$ is present, then all those vertices have a cost of $d + 2$ except $c^i_j$ that has a cost $d + 3$ or $d + 2$ depending on whether $e_i$ is eliminated or not yet.

Overall when we eliminate $A$, we pay a marginal cost of $(k - E_s) + R_n$ where $E_s$ is the number of sets eliminated and $R_n$ is the number of integers not yet eliminated. The marginal cost of the order $\prec$ is at least the marginal cost of all vertices eliminated before $A$ plus the marginal cost for $A$. Because the marginal cost of vertices removed before $A$ is at least $E_s$, by adding the marginal cost of $A$ we get a marginal cost larger than $E_s + (k - E_s) + R_n = k + R_n$ which can be equal to $k$ only if $R_n = 0$ which means that all vertices $e_i$ corresponding to integers $\{1 \ldots n\}$ have been eliminated. Note that if a vertex $e_i$ is directly eliminated without eliminating first a vertex $s_j$ and a triplet $a^i_j$, $b^i_j$, $c^i_j$ then we have to add a marginal cost of 1 specifically for this vertex $e_i$. But in that case, it means that the marginal cost of all vertices before $A$ includes the cost of removing this $e_i$ which means that we cannot have an overall marginal cost of $k$. Combining everything we get that if we have an order that has a marginal cost of $k$ and thus a cost of at most $V$, then we have $k$ sets $S_{i_1}, \ldots, S_{i_k}$ covering all integers in $U$. 