Jagiellonian University
Faculty of Mathematics and Computer Science
university major: Computational mathematics

Bachelor of Science Thesis

Comparative Analysis of Graph Sparsification Methods in Shortest Path Algorithm Optimization

Aleksandra Kowalska

Academic Advisor
dr Edward Szczypka

June 20, 2025
Kraków, Poland

Contents

1 Introduction

This work is meant to address the problem of optimizing the shortest path algorithm within the context of graph sparsification. The topic constitutes a significant area within graph theory and finds extensive applications in algorithm optimization, computer networks, and the analysis of large datasets. The primary objective of sparsification is to reduce the number of edges in a graph while preserving its essential properties, including the correctness of computed shortest paths. In this regard, methods that guarantee the maintenance of key graph parameters, even after a substantial reduction in the number of edges, play a particularly crucial role.

Some graph reduction methods rely on heuristics, while others provide formal guarantees for preserving key graph properties. The aim of this paper is to present and analyze various sparsification techniques and their impact on the efficiency of shortest path algorithms.

The work is divided into chapters. The first two provide an introduction to graph theory, along with fundamental definitions and notations used throughout the remainder of the work, as well as metrics playing a key role in graph analysis and comparison. The third and fourth chapters will discuss sparsification methods as well as a theoretical discussion regarding their computational complexity. The few final sections will present an experimental analysis of selected graph reduction methods, including the impact of the number of removed edges on the quality and changes in computational efficiency of previously discussed metrics.

The selection of this research topic is largely motivated by the author's personal interest in issues related to the computational optimization of algorithms using mathematical tools. The problem of graph reduction is an area that combines both theoretical and practical aspects — on one hand, it is based on mathematical foundations, and on the other, it has wide applicability in real-world problems involving the processing of large datasets. A comparative analysis of different graph sparsification methods in the context of the shortest path problem allows for not only the assessment of their computational efficiency and the quality of the obtained results, but also the identification of potential directions for further considerations in the field of designing graph structures optimized for specific algorithmic applications.

This work aims to present existing solutions as well as explore novel approaches to graph reduction and their influence on state-of-the-art algorithms.

2 Preliminaries

In this chapter we will introduce key concepts as well as notation used throughout this work.

2.1 Basic definitions

We define a *graph* as a pair $G = (V, E)$, where V is a finite, nonempty set of vertices, and E is a set of edges.

- In an *undirected graph*, $E \subset \binom{V}{2}$ consists of unordered pairs $\{u, v\}$ with $u \neq v$, representing bidirectional connections.
- In a *directed graph*, $E \subset V \times V$ consists of ordered pairs (u, v) with $u \neq v$, where each edge has a direction from u (the *tail*) to v (the *head*).

In the case of a *weighted graph*, we additionally introduce a function $w : E \to \mathbb{R}_+$ that assigns a non-negative weight $w(e)$ to each edge $e \in E$.

A *subgraph* of a graph $G = (V, E, w)$ is a graph $H = (V, E', w_{|E'})$ such that $E' \subset E$. The common notation is $H \subset G$.

A *walk* in a graph G is a sequence of vertices $v_1, \ldots, v_k$ such that $(v_i, v_{i+1}) \in E$ for all $i = 1, \ldots, k-1$.

A *path* in a graph G is a walk $v_1, \ldots, v_k$ in which all vertices are pairwise distinct.

We say that a graph $G = (V, E)$ is *connected* if for any pair of vertices $u, v \in V$ there exists a walk in G from u to v.

A *cycle* in a graph G is a sequence of vertices $v_1, v_2, \ldots, v_k$ with $k \geq 3$ such that:

- $v_1 = v_k$ (the cycle closes),
- all edges $(v_i, v_{i+1}) \in E$ for $i = 1, \ldots, k-1$ are distinct,
- and all vertices $v_1, \ldots, v_{k-1}$ are pairwise distinct.

A graph is called *acyclic* if it contains no cycles.

If for some $u, v \in V$ it holds that $(u, v) \in E$, we say that vertex u is a *neighbor* of vertex v. The set of all neighbors of a given vertex u is typically denoted by $N(u)$.

The *degree* of a vertex $u \in V$ in an undirected graph is the number of its neighbors, denoted by $\deg(u)$, where $\deg(u) = |\{v \in V : (u, v) \in E\}|$. In directed graphs, a distinction is made between edges entering and exiting a given vertex. Thus, for each $u \in V$, we define:

- *in-degree* $\deg^-(u)$, being the number of edges for whom u is the head vertex,
- *out-degree* $\deg^+(u)$, being the number of edges for whom u is the tail vertex.

Additionally, for weighted graphs we define the *weighted node degree* as the sum of the edge weights for out-going edges incident to that node. In situations where $\deg(u) = 0$, we say that vertex u is *isolated*.

For a weighted directed graph $G = (V, E, w)$, an *adjacency matrix* $A = (a_{ij}) \in \mathbb{R}^{|V| \times |V|}$ is specified as:

$$A_{ij} = \begin{cases} w(v_i, v_j) & \text{if } (v_i, v_j) \in E, \\ 0 & \text{otherwise.} \end{cases}$$

For unweighted graphs it holds that $\forall (v_i, v_j) \in E : w(v_i, v_j) = 1$.

The *weighted degree matrix* $D = (d_{ij}) \in \mathbb{R}^{|V| \times |V|}$ is a diagonal matrix where

$$D_{ij} = \begin{cases} \sum_{(v_i, v_k) \in E} w(v_i, v_k) & \text{if } i = j, \\ 0 & \text{if } i \neq j. \end{cases}$$

A *spanning tree* of a connected, undirected graph $G = (V, E)$ is a subgraph $T = (V, E_T)$ which is a *tree* (i.e., connected and acyclic graph) and includes all vertices of G.

A *minimum spanning tree* (MST) of a weighted, undirected graph is a spanning tree T that minimizes the total weight $\sum_{e \in E_T} w(e)$.

2.2 Graph sparsification

We will now establish a proper definition of the graph sparsification process.

Let $G = (V, E, w)$ be a weighted directed graph. A *sparsified* subgraph $H = (V, E', w_{|E'})$ of $G = (V, E, w)$ is a graph such that

$$|E'| = (1 - \rho)|E|,$$

where ρ is a set value from $[0, 1]$. We will call the map

$$f : G \to H, \quad H = f(G)$$

a *graph sparsification algorithm*, or *sparsifier* for short, and ρ the *prune rate*.

There is a wide plethora of sparsification algorithms, each with varying objectives regarding the preservation of graph properties, and of different complexities. Some of them perform vertex sparsification, while others focus on edge reduction. This study will be confined to edge sparsification, meaning that the original vertices will always remain unchanged and only the graph's edge set will be the one to undergo transformation. This approach is adopted because the majority of graph metrics require the complete vertex set in order to evaluate and compare the performance of sparsification algorithms. Moreover, it is important to note that in graphs of practical significance — particularly in real-world applications — the edge set typically outnumbers the vertex set, thus containing a greater volume of potentially redundant information.

3 Important metrics

The rest of this work aims to focus on the most promising graph sparsification algorithms within the context of the research focus. By a "promising algorithm," we refer to a sparsifier that reduces the total number of edges while simultaneously striving to preserve the structural and functional properties of a graph as best as possible. This naturally requires laying down some fundamental metrics, which will allow for a more thorough analysis and interpretation of the sparsifiers' impact on the graph.

3.1 General metrics

Laplacian Matrix for Directed and Symmetrized Graphs

The Laplacian matrix — and its quadratic form in particular — quantifies the smoothness of the function f on the graph: the smaller $Q(f)$, the more similar the values of f are on adjacent vertices.

Let $G = (V, E, w)$ be a directed weighted graph. We now consider two versions of the Laplacian matrix.

Directed Laplacian (non-symmetric). For a directed graph with n vertices, one can define the *(out-degree) Laplacian operator* as:

$$L^{\text{dir}} := D^{\text{out}} - A^{\text{dir}},$$

where $A^{\text{dir}} \in \mathbb{R}^{n \times n}$ is the asymmetric adjacency matrix with

$$A^{\text{dir}}_{ij} = \begin{cases} w(v_i, v_j) & \text{if } (v_i, v_j) \in E, \\ 0 & \text{otherwise.} \end{cases}$$

and D^{out} is the diagonal matrix of out-degrees:

$$D^{\text{out}}_{ii} = \sum_{j=1}^{n} A^{\text{dir}}_{ij}.$$

Symmetrized Graph and Laplacian. To enable symmetric analysis, we define the *symmetrized version* of a directed graph G as an undirected graph $G^{\text{sym}} = (V, E^{\text{sym}}, w^{\text{sym}})$, where:

$$E^{\text{sym}} = \{\{u, v\} : (u, v) \in E \text{ or } (v, u) \in E\},$$

and the symmetric weight function is defined as:

$$w^{\text{sym}}(u, v) = \begin{cases} w(u, v) + w(v, u) & \text{if both edges exist,} \\ w(u, v) & \text{if only } (u, v) \in E, \\ w(v, u) & \text{if only } (v, u) \in E. \end{cases}$$

Then the (combinatorial) Laplacian of G^{sym} is given by:

$$L := D - A,$$

where:

$$A_{ij} = w^{\text{sym}}(v_i, v_j), \quad D_{ii} = \sum_{j=1}^{n} A_{ij}$$

Laplacian Quadratic Form

Let $f : V \to \mathbb{R}$ be a scalar function on the vertices of G, identified with a vector $f \in \mathbb{R}^n$ under a fixed ordering. The *Laplacian quadratic form* associated with the symmetrized Laplacian L is defined by

$$Q(f) := f^\top L f.$$

This can be expanded as:

$$\begin{aligned} Q(f) &= f^\top D f - f^\top A f \\ &= \sum_{i=1}^{n} D_{ii} f_i^2 - \sum_{i,j=1}^{n} A_{ij} f_i f_j \\ &= \sum_{\{u,v\} \in E^{\text{sym}}} w^{\text{sym}}(u,v) \left(f_i^2 + f_j^2 - 2 f_i f_j \right) \\ &= \sum_{\{u,v\} \in E^{\text{sym}}} w^{\text{sym}}(u,v) (f(u) - f(v))^2. \end{aligned}$$

Thus $Q(f) = \sum_{(u,v) \in E} w^{\text{sym}}(u,v)(f(u) - f(v))^2$.

Example 1 (Symmetrized Laplacian and Quadratic Form) *Let $G = (V, E, w)$ be a directed weighted graph with*

$$V = \{v_1, v_2, v_3\}, \quad E = \{(v_1, v_2), (v_2, v_3)\}, \quad w(v_1, v_2) = 2, \quad w(v_2, v_3) = 3$$

The symmetrized graph has:

$$E^{\text{sym}} = \{\{v_1, v_2\}, \{v_2, v_3\}\}, \quad w^{\text{sym}}(v_1, v_2) = 2, \quad w^{\text{sym}}(v_2, v_3) = 3$$

Then:

$$A = \begin{bmatrix} 0 & 2 & 0 \\ 2 & 0 & 3 \\ 0 & 3 & 0 \end{bmatrix}, \quad D = \begin{bmatrix} 2 & 0 & 0 \\ 0 & 5 & 0 \\ 0 & 0 & 3 \end{bmatrix}, \quad L = D - A = \begin{bmatrix} 2 & -2 & 0 \\ -2 & 5 & -3 \\ 0 & -3 & 3 \end{bmatrix}.$$

For $f = \begin{bmatrix} 1 \\ 2 \\ 4 \end{bmatrix}$, the quadratic form is:

$$Q(f) = f^\top L f = (1-2)^2 \cdot 2 + (2-4)^2 \cdot 3 = 2(1)^2 + 3(2)^2 = 2 + 12 = 14$$

Degree distribution

This distribution allows for the analysis of how sparsification affects the local structure of the graph, particularly with respect to the behavior of high-degree vertices.

The *vertex degree distribution* of a graph is the probability distribution of the random variable D_G, which for each vertex $v \in V$ takes the value $\deg(v)$. For each $k \geq 0$, we define:

$$P(D_G = k) = \frac{|\{v \in V : \deg_G(v) = k\}|}{|V|}.$$

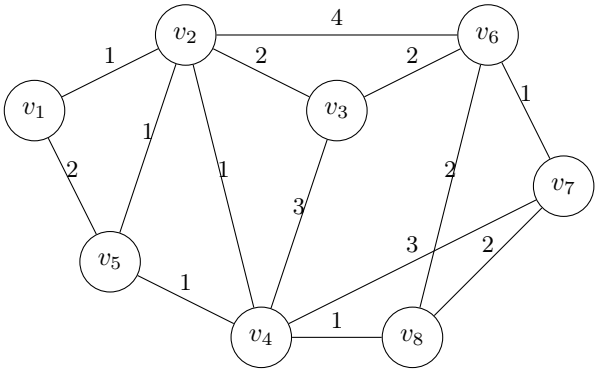
Example 2 (Degree distribution) *Consider the following graph with 8 vertices and 13 edges. The graph is weighted but edge weights play no role in the degree-based distribution calculations. The degrees of vertices are as follows.*

- $\deg(v_1) = 2$
- $\deg(v_3) = \deg(v_5) = \deg(v_7) = \deg(v_8) = 3$

- $\deg(v_6) = 4$
- $\deg(v_2) = \deg(v_4) = 5$

Degree distributions:

$$P(D_G = 2) = \frac{1}{8}$$
$$P(D_G = 3) = \frac{4}{8} = 0.5$$
$$P(D_G = 4) = \frac{1}{8}$$
$$P(D_G = 5) = \frac{2}{8} = 0.25$$



3.2 Distance metrics

Let $G = (V, E, w)$ be a connected weighted graph, where $w : E \to \mathbb{R}_+$ is a weight function assigning positive weights to the edges. For any pair of vertices $u, v \in V$, the *distance* $d_G(u, v)$ is defined as the minimum total weight of any path connecting them:

$$d_G(u, v) = \min_{\pi \in \Pi_{u,v}} \sum_{e \in \pi} w(e),$$

where $\Pi_{u,v}$ denotes the set of all paths from u to v, and each path π is viewed as a sequence of consecutive edges. If no such path exists, we define $d_G(u, v) := \infty$. For a sparsified subgraph $H = (V, E', w_{|E'}) \subseteq G$, distances $d_H(u, v)$ are defined analogously, using $w_{|E'}$ in place of w.

To precisely analyze the changes in shortest path distances caused by a given sparsification algorithm, we will utilize the following metrics.

Graph diameter

The diameter is a key metric in evaluating communication efficiency in a graph and finds widespread application in the design of communication networks and the analysis of computer networks.

The *diameter* of a graph is defined as the maximum distance between any pair of its vertices. Formally, it is the smallest number s such that any two vertices are connected by a path of length at most s:

$$\operatorname{diam}(G) = \begin{cases} \max_{u,v \in V} d_G(u, v) & \text{if } G \text{ is a connected graph,} \\ \infty & \text{otherwise.} \end{cases}$$

Unreachable pairs ratio

This metric gives meaningful insight to the overall vertex accessibility after the sparsification process.

The *unreachable pairs ratio* is defined as the ratio of the number of vertex pairs that are mutually unreachable in H to the total number of vertex pairs in G:

$$U(H) = \frac{|\{(u,v) \in V \times V : d_H(u,v) = \infty\}|}{|V|(|V|-1)}.$$

All-Pairs Shortest Path (APSP)

This metric captures the minimal distance between any pair of vertices consisting of a source vertex $u \in V$ and a target vertex $v \in V$. To compute APSP distances, we will employ classic graph algorithms, in particular:

- Johnson's algorithm — efficient for sparse graphs with non-negative edge weights
- Floyd-Warshall algorithm — suitable for denser graphs or graphs with unknown structure

The output of this computation consists of distance matrices $D_G = [d_G(u,v)]$ and $D_H = [d_H(u,v)]$, which will be compared to assess the accuracy of distance approximation in the sparsified graph.

Other Path Parameters

These parameters are intended to provide a more comprehensive picture of the structural integrity preserved by the sparsified graph H relative to the original graph G — in particular, to identify sparsifiers that best preserve the structure of shortest paths.

1. *Local Path Stretch Factor*:

$$\sigma_H(u,v) = \begin{cases} \frac{d_H(u,v)}{d_G(u,v)} & \text{if } (u,v) \in E' \text{ and } (v,u) \in E', \\ \infty & \text{otherwise.} \end{cases}$$

i.e., the ratio of the distance between vertices u and v in the sparsified graph H to the corresponding distance in the original graph G.

2. *Average Relative Path Stretch*:

$$\overline{\sigma}_H = \frac{1}{|\mathcal{P}|} \sum_{(u,v)\in\mathcal{P}} \sigma_H(u,v),$$

where $\mathcal{P} = \{(u,v) \in V \times V : d_G(u,v) < \infty\}$ is the set of all mutually reachable vertex pairs in G.

3. *Variance of the Path Stretch Factor*:

$$\mathrm{Var}[\sigma_H] = \frac{1}{|\mathcal{P}|} \sum_{(u,v)\in\mathcal{P}} \left(\sigma_H(u,v) - \overline{\sigma}_H\right)^2.$$

This metric allows us to assess the stability of path lengths after sparsification.

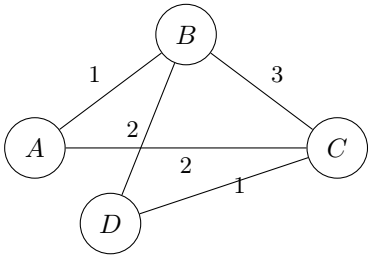
4. *Maximum Stretch Factor*:

$$\sigma_H^{\max} = \max_{(u,v)\in\mathcal{P}} \sigma_H(u,v),$$

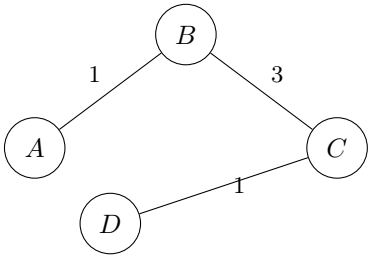
i.e. the worst-case impact of sparsification on path lengths.

Example 3 (Path stretch parameters) *Consider the following weighted graph G with 4 vertices and 5 edges, and its sparsified subgraph H in which shortest paths are elongated.*

Graph G



Sparsified subgraph H



Shortest path distances in G

$d_G(u,v)$	A	B	C	D
A	0	1	2	3
B	1	0	3	2
C	2	3	0	1
D	3	2	1	0

Shortest path distances in H

$d_H(u,v)$	A	B	C	D
A	0	1	4	5
B	1	0	3	4
C	4	3	0	1
D	5	4	1	0

For all pairs $u \neq v$, $\sigma_H(u,v) = \frac{d_H(u,v)}{d_G(u,v)}$, and so we have

$$\sigma_H(A,B) = \frac{1}{1} = 1 \qquad \sigma_H(B,C) = \frac{3}{3} = 1$$
$$\sigma_H(A,C) = \frac{4}{2} = 2 \qquad \sigma_H(B,D) = \frac{4}{2} = 2$$
$$\sigma_H(A,D) = \frac{5}{3} \approx 1.67 \qquad \sigma_H(C,D) = \frac{1}{1} = 1$$

Average path stretch:

$$\sigma_H = \frac{1}{6}\left(1+2+1.67+1+2+1\right) \approx \frac{8.67}{6} \approx 1.44$$

Variance:

$$\mathrm{Var}[\sigma_H] = \frac{1}{6}\left[(1-1.44)^2 + (2-1.44)^2 + (1.67-1.44)^2 + (1-1.44)^2 + (2-1.44)^2 + (1-1.44)^2\right] \approx 0.22$$

Maximum stretch factor:

$$\sigma_H^{\max} = 2$$

4 Graph sparsification algorithms

This chapter is dedicated to the definition and characterization of graph sparsification algorithms.

Let $G = (V, E, w)$ be a graph, and $H = (V, E', w_{|E'})$ a subgraph of G.

4.1 Random sparsifier

The random sparsifier method constructs the graph H by choosing to keep every edge $e \in E$ with equal probability $\rho \in [0, 1]$.

This procedure is fairly simple in regard to implementation and is easily scalable, however, it cannot be guaranteed to maintain the original graph's structural properties or integrity.

4.2 K-Neighbor sparsifier

In this method, for every vertex $v \in V$, exactly k edges from the set $\{(v, u) : u \in N(v)\}$ are maintained, provided that $\deg(v) \geq k$ (for directed graphs it must be specified whether it is $\deg^+(v)$ or $\deg^-(v)$ being considered). Otherwise, if $\deg(v) \leq k$, all edges whose tail is v are kept in H.

Should there be a need to choose which k edges are to be preserved, the edges are selected with probabilities proportional to their weight, meaning

$$\mathbb{P}(e_{v_i} \in E') = \frac{w_{v_i}}{\sum_{u \in N(v)} w_{vu}}.$$

This method assures that every vertex in H has degree of at least 1, contributing to the maintenance of the initial graph's connectivity.

4.3 Local Degree sparsifier

This method constructs the set of edges for H by choosing for every vertex $v \in V$ a subset of its edges leading to $\lfloor d(v)^\rho \rfloor$ neighbors of v, where neighbors are chosen by their degree in descending order.

The goal of this method is to preserve edges leading to high-degree vertices, which play a key role in the network structure of the graph.

4.4 t-Spanner

This algorithm constructs H by iteratively adding edges from the original graph G to H, starting with the edge of the smallest weight. An edge $(u, v) \in E$ with weight w is added to H only if the shortest path between vertices u and v in the current graph H is longer than $t \cdot w$, where $t \geq 1$ is a given parameter determining the spanner ratio.

The goal of this method is to create a subgraph where for any pair of vertices $u, v \in V$, the shortest path distance in H is preserved up to a factor of at most t compared to the shortest path distance in G, i.e., the following condition holds:

$$\forall u, v \in V : d_H(u, v) \leq t \cdot d_G(u, v).$$

For directed graphs, the input graph G is first symmetrized (ignoring edge directions to determine distances and select edges). The edges are then added to H while retaining their original direction.

4.5 MST

The subgraph $H = (V, E', w_{|E'})$ is constructed by generating a Minimum Spanning Tree (MST) of G. The set

$$E' = \{ e_1, e_2, \ldots, e_{|V|-1} \}$$

is a set of edges selected by Kruskal's algorithm. The resulting MST contains $|V| - 1$ edges and preserves the original graph's connectivity.

4.6 KOLS Sparsifier

This heuristic is based on multiple iterations of the Breadth-First Search (BFS) algorithm performed on the graph $G = (V, E, w)$. The priority of an edge is determined by the frequency with which it appears as a tree edge in k independent BFS traversals, each starting from a randomly chosen vertex.

We define the following parameters:

- $k \in \mathbb{N}$ – the number of BFS runs
- $\rho \in (0, 1]$ – the desired prune rate

Let $f(u, v)$ denote the number of times the edge $(u, v) \in E$ has appeared during any BFS traversal.

To ensure the resulting sparsified graph is connected, the algorithm first constructs a spanning tree T_0 of G using a BFS traversal from the first randomly chosen root. Then, it performs $k - 1$ BFS traversals from pairwise distinct random roots (different from the one used for T_0), incrementing $f(u, v)$ for each edge traversed as a tree edge.

After collecting the frequencies, all edges are sorted in descending order by $f(u, v)$. The final sparsified edge set $E' \subseteq E$ is constructed as follows:

1. T_0 is included in E' to guarantee connectivity
2. The remaining
$$m' = \max\{|V| - 1, \lfloor \rho \cdot |E| \rfloor\} - |T_0|$$
edges with the highest $f(u, v)$ among $E \setminus T_0$ are added to E'

The resulting sparsified graph is $H = (V, E', w_{|E'})$, where the weight of each retained edge is the original weight.

This particular algorithm places special emphasis on preserving the connectivity structure of the original graph. This stems from the inherent nature of the BFS algorithm, which prioritizes traversal along bridge-like edges — those essential for reaching unexplored regions of the graph. As a result, edges that serve as critical connections between components are likely to appear frequently across multiple BFS runs and are thus favored during edge selection. By incorporating an initial spanning tree and reinforcing it with high-frequency edges, it makes this heuristic somewhat well-suited for scenarios where maintaining global graph connectivity is a priority.

Note 1 *If T_0 is omitted, the algorithm simply returns the top m' edges by frequency, but the resulting subgraph may be disconnected.*

Additional remarks. This algorithm is subject to further optimization by introducing a post-processing phase in which the spanning tree T_0 is iteratively improved. Specifically, for each edge $e \in E \setminus T_0$ such that $f(e)$ is greater than the minimum frequency among edges in T_0, we can consider replacing an edge $e' \in T_0$ (with $f(e') < f(e)$) by e, provided that the resulting edge set remains acyclic and spans V. In other words, this corresponds to performing edge swaps $T_0 \leftarrow T_0 \cup \{e\} \setminus \{e'\}$ such that T_0 remains a spanning tree after each swap. This exchange process can potentially maximize the number of key edges within the spanning structure, while preserving global connectivity.

5 Comparison of graph reduction algorithms

Table 1: Sparsifiers' applicability to types of graphs and their characteristics

Sparsifier	Directed?	Weighted?	Deterministic?	Average complexity**	Worst-case complexity**
Random	✓	✓	✗	$O(\|E\|)$	$O(\|E\|)$
K-Neighbor	✓*	✓	✗	$O(\|V\|^2)$	$O(\|E\| + \|V\|^{1+\rho})$
Local Degree	✓*	✓	✓	$O(\|V\| \log \|V\|)$	$O(\|V\|^2 \log \|V\|)$
t-Spanner	✗	✓	✓	$O(\|E\|^4 \log \|V\|)$	$O(\|E\|^4 \log \|V\|)$
MST	✗	✗	✗	$O(\|E\| \log \|V\|)$	$O(\|E\| \log \|V\|)$
KOLS	✓	✓	✗	$O(\|V\|^2 \log \|V\|)$	$O(\|V\|^2 \log \|V\|)$

* Need to specify using in-degree or out-degree.
** For dense graphs.

It is important to emphasize that, given that not all sparsifiers are designed to operate on directed graphs, any directed input graph underwent a symmetrization procedure. This process involves adding a reciprocal [head, tail] edge to the graph, provided that such an edge does not already exist. This step ensures the proper functioning of the algorithm and guarantees that all subsequent calculations and metric evaluations can be performed.

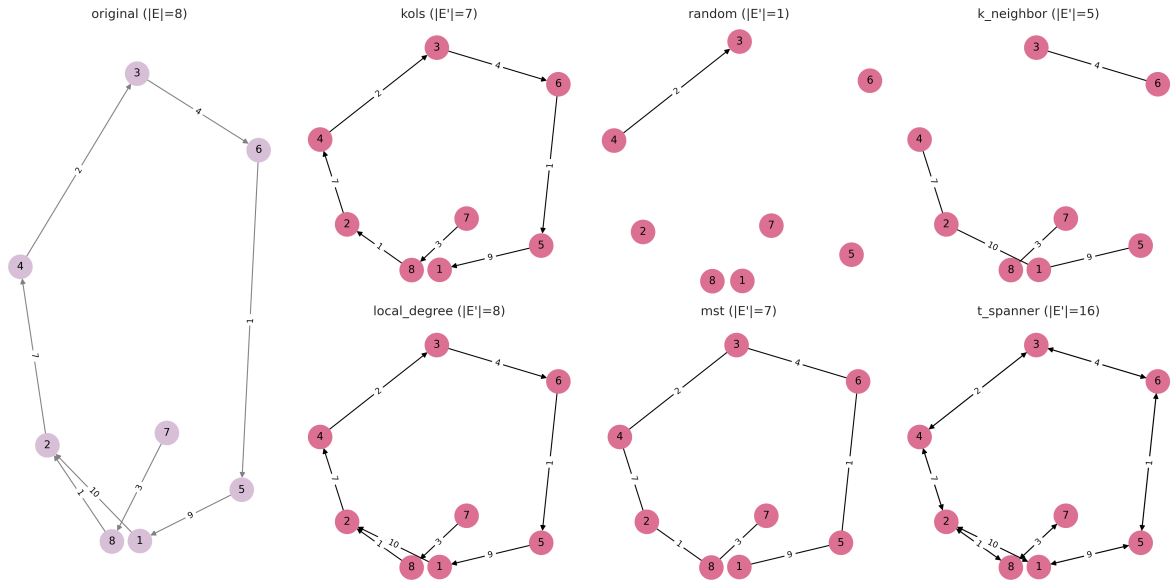


Figure 1: Comparison of graph sparsification algorithms for an example directed weighted graph.

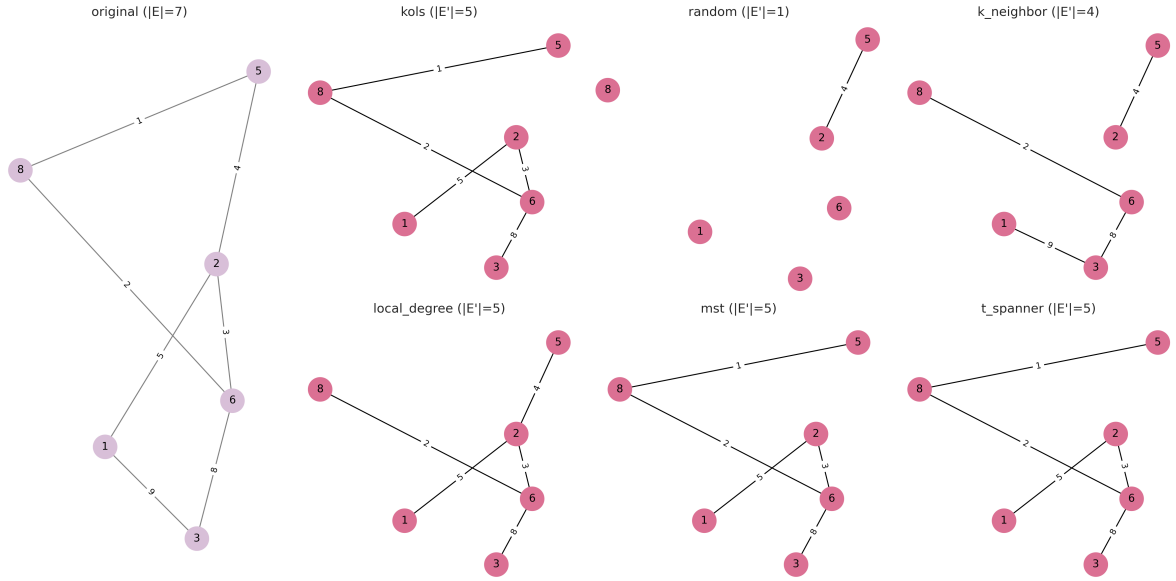


Figure 2: Comparison of graph sparsification algorithms for an example undirected weighted graph.

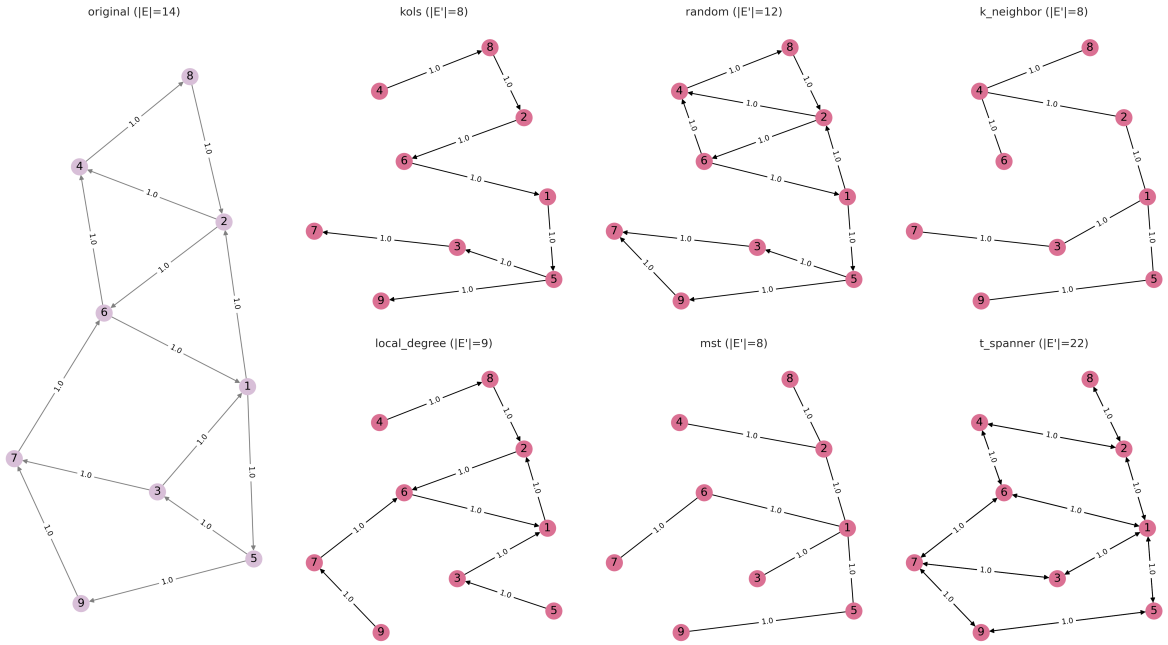


Figure 3: Comparison of graph sparsification algorithms for an example directed unweighted graph.

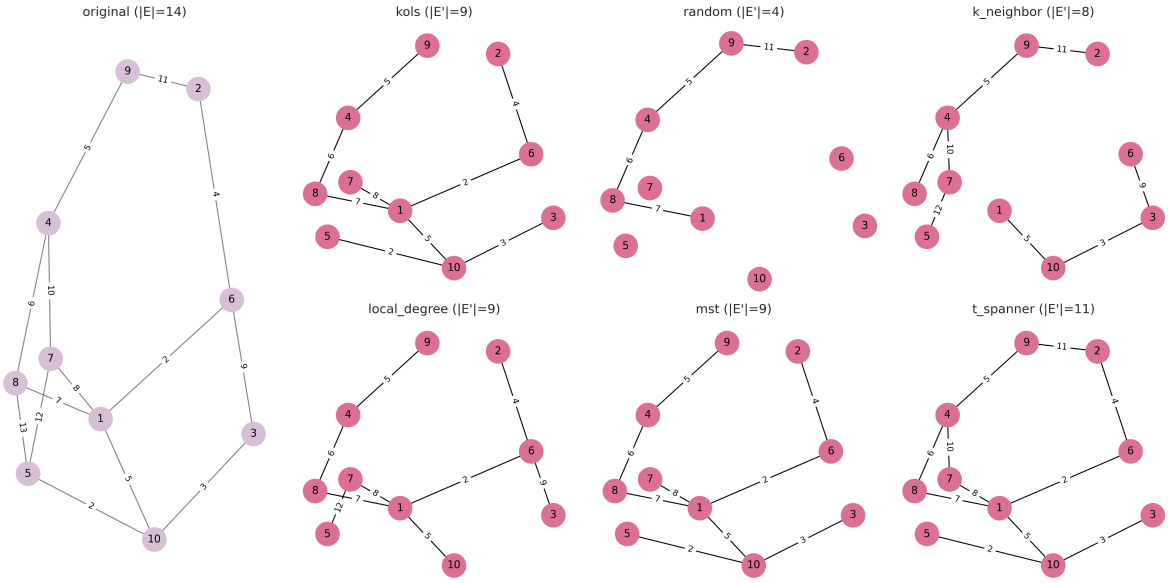


Figure 4: Comparison of graph sparsification algorithms for an example undirected weighted bipartite graph.

6 Pseudocode and theoretical complexity analysis

For ease of notation in further calculations, for a graph $G = (V, E, w)$ we denote $|V| =: n$, $|E| =: m$, and for the sparsified subgraph $|E'| =: m'$. We also assume that every input graph is connected.

Note 2 *For a simple connected graph, we have $n - 1 \leq m' < n^2$. Thus, replacing $\log_2 m'$ with $\log_2 n$ in asymptotic expressions like $m' \log_2 m'$ does not affect the Θ-class. The same holds for directed graphs after symmetrization, up to an additive constant.*

6.1 Random

Algorithm 1 Random Sparsifier

```
1: procedure RandomSparsify(G, ρ)
2:     Input: Graph G = (V, E, w), probability ρ ∈ [0, 1]
3:     Output: Sparsified subgraph H = (V, E', w_|E')
4:
5:     for each edge e ∈ E do
6:         if random() < ρ then          ▷ random() returns a uniformly distributed value in [0, 1]
7:             Add e to E'
8:
9:     Return: H = (V, E', w_|E')
```

Complexity analysis

Pessimistic running time. The procedure inspects every edge exactly once and performs $O(1)$ work on it (we can assume that primitive operations such as a comparison with `random()`, pushing one element to a list, or drawing a value from `RandomChoices` run in $O(1)$ time). Hence, in the deterministic sense:

$$T_W(G) = \sum_{e \in E} \Theta(1) = \Theta(m).$$

Expected running time. Let us assume the following notation:

- X_e - indicator that edge $e \in E$ is kept, where $\mathbb{P}[X_e = 1] = \rho$
- c_0 - cost of the deterministic work of the algorithm (operations like reading the edge and random number generation)
- c_1 - extra cost of inserting an edge into E'

The random runtime variable is

$$T(G) = \sum_{e \in E} (c_0 + c_1 X_e).$$

Taking the expected value and using linearity we obtain

$$\mathbb{E}[T(G)] = \sum_{e \in E} (c_0 + c_1 \mathbb{E}[X_e]) = (c_0 + c_1 \rho) m = \Theta(m).$$

Thus both average and and worst-case complexities are equal to $\Theta(m)$.

6.2 K-Neighbor

Algorithm 2 k-Neighbour Sparsifier

procedure kNeighbourSparsify(G, ρ)
  Input: Weighted graph $G = (V, E, w)$, exponent $\rho \in [0, 1]$
  Output: Sparsified subgraph $H = (V, E', w_{|E'})$

  for each vertex $v \in V$ **do**
    $E_v \leftarrow N(v)$
    $d_v \leftarrow |E_v|$
    $k_v \leftarrow \max\{1, \lfloor d_v^\rho \rfloor\}$

    if $d_v \leq k_v$ **then**
      $E' \leftarrow E' \cup E_v$ ▷ keep all outgoing edges from v
    else
      $\forall u \in N(v) : \ \mathbb{P}(vu) = \dfrac{w(v,u)}{\sum_{x \in N(v)} w(v,x)}$
      $\text{Selected}_v \leftarrow k_v$ randomly sampled distinct edges from E_v according to $\mathbb{P}(vu)$
      $E' \leftarrow E' \cup \text{Selected}_v$

  return $H = (V, E', w_{|E'})$

Complexity analysis

Let $d_v = \deg^+(v)$ and $\Delta = \max_v d_v \leq n - 1$. Define

$$k_v \coloneqq \max(1, \lfloor d_v^\rho \rfloor) \text{ for } 0 \leq \rho \leq 1.$$

The per-vertex work requires:

1. Scanning its adjacency list to compute totals and build a cumulative distribution — $\Theta(d_v)$
2. Drawing k_v edges without replacement according to the weights — $\Theta(k_v)$

Hence

$$T(G) = \Theta\Big(\sum_{v \in V} (d_v + k_v)\Big) = \Theta\Big(m + \sum_{v \in V} k_v\Big).$$

Pessimistic running time. Because $k_v \leq 1 + d_v^\rho$, we can bound the drawing term:

$$\sum_{v \in V} k_v \leq n + \sum_{v \in V} d_v^\rho \leq n + n\Delta^\rho = \Theta(n^{1+\rho}),$$

so

$$T_W(G) = \Theta(m + n^{1+\rho}).$$

Expected running time. We can model the input as $G(n,p)$ with $p = \frac{m}{\binom{n}{2}}$. Then $D \sim \mathrm{Bin}(n-1,p)$, where D is the out-degree of a fixed vertex. Because $x \mapsto x^\rho$ is concave on $[0,\infty)$ when $0 \le \rho \le 1$, Jensen's inequality yields

$$\mathbb{E}[D^\rho] \;\le\; \big(\mathbb{E}[D]\big)^\rho \;=\; \big((n-1)p\big)^\rho.$$

Therefore

$$\mathbb{E}\left[\sum_v d_v^\rho\right] \le n^{1+\rho}p^\rho$$

and

$$\mathbb{E}[T(G)] = \Theta(m + n^{1+\rho}p^\rho + n).$$

- **Dense graphs** $p = \Theta(1)$ $(m = \Theta(n^2))$: $T = \Theta(n^2)$.
- **Sparse graphs** $p = \Theta(1/n)$ $(m = \Theta(n))$: $T = \Theta(n^{1+\rho})$.

The lower bound $\Omega(m)$ is immediate because every edge is examined at least once.

6.3 Local Degree

Algorithm 3 Local Degree Sparsifier

```
procedure LocalDegreeSparsify(G, ρ)
    Input: Graph G = (V, E, w), exponent ρ ∈ [0, 1]
    Output: Sparsified subgraph H = (V, E', w_|E')

    for each v ∈ V do
        D[v] ← deg⁺(v)

    for each v ∈ V do
        k_v ← ⌊D[v]^ρ⌋                          ▷ number of neighbors to select for vertex v
        N_v ← N(v)
        Sort N_v in descending order based on D[u] for u ∈ N_v

        if k_v > 0 then
            Select the first k_v neighbors from the sorted N_v

            for each u in the selected k_v neighbors do
                E' ← E' ∪ (v, u)

    Return: H = (V, E', w_|E')
```

Complexity analysis

Let $d_v = \deg^+(v)$ and $\Delta = \max_v d_v$. Adjacency lists are stored together with their length, so reading d_v is constant time, but if the implementation required computing every individual degree, the complexity of generating the degree table would take $\Theta(n+m)$ instead of $\Theta(n)$. This however does not change any

asymptotic bound below. We also assume a comparison-based sorting algorithm with a log-linear cost. Define:

$$k_v \coloneqq \lfloor d_v^\rho \rfloor \qquad \text{for } 0 \le \rho \le 1.$$

The first step is computing the degree table, which, as discussed before, takes linear time

$$T_1 = \sum_{v \in V} \Theta(1) = \Theta(n).$$

Next, the local processing for a fixed vertex v consists of two steps:

1. Sorting its neighborhood — $\Theta(d_v \log d_v)$
2. Copying the first k_v edges — $\Theta(k_v)$

Hence

$$T_2 = \sum_{v \in V} (\Theta(d_v \log d_v) + \Theta(k_v)). \tag{1}$$

Because $d_v \le \Delta \le n-1$, we can bound the sorting term:

$$d_v \log d_v \le d_v \log n \;\Rightarrow\; \sum_{v \in V} d_v \log d_v \le \log n \sum_{v \in V} d_v \le 2m \log n = \Theta(m \log n). \tag{2}$$

Similarly, since $k_v \le d_v^\rho$, the copying term can be bounded by:

$$\sum_{v \in V} k_v \le \sum_{v \in V} d_v^\rho. \tag{3}$$

Pessimistic running time. For the worst-case bound (since $0 \le \rho \le 1$) we can write

$$\sum_{v \in V} d_v^\rho \le \sum_{v \in V} d_v \le m. \tag{4}$$

Note 3 *For a directed graph, $\sum_v d_v = m$. The factor 2 applies only in the undirected case but the distinction is immaterial for the asymptotic bounds.*

Inserting (2), (3) and (4) into (1) gives

$$T_W(G) = \Theta(m \log n) = \Theta(m \log n). \tag{5}$$

The lower bound $\Omega(m)$ holds because every edge is inspected at least once; thus (5) is asymptotically tight for both dense and sparse graphs.

Expected running time. For the expected bound with $G \sim G(n,p)$, the out-degree $D \sim \mathrm{Bin}(n-1,p)$ and Jensen's inequality (on concave x^ρ) yields

$$\mathbb{E}[D^\rho] \le ((n-1)p)^\rho,$$

therefore

$$\mathbb{E}\left[\sum_v d_v^\rho\right] \leq n((n-1)p)^\rho = \Theta(n^{1+\rho}p^\rho). \tag{6}$$

Using (2) with $\mathbb{E}[d_v] = (n-1)p$ and (6):

$$\mathbb{E}[T(G)] = \Theta(m \log n + n^{1+\rho}p^\rho + n) \tag{7}$$

- **Dense graphs** $p = \Theta(1)$ ($m = \Theta(n^2)$): the dominating term in (7) becomes $\Theta(n^2 \log n)$.
- **Sparse graphs** $p = \Theta(1/n)$ ($m = \Theta(n)$): (7) becomes $\Theta(n \log n)$.

6.4 t-Spanner

Algorithm 4 t-Spanner Sparsifier

procedure t-SpannerSparsify(G, t)
  Input: Weighted graph $G = (V, E, w)$, stretch factor $t \geq 1$
  Output: Sparsified subgraph $H = (V, E', w_{|E'})$

  if G is directed **then**
    $G_{\text{sym}} \leftarrow$ SymmetrizeGraph(G)
    $G \leftarrow G_{\text{sym}}$ ▷ use symmetrized graph for processing

  Sort all edges in E by their weights in non-decreasing order: E_{sorted}

  for each edge $(u, v, w) \in E_{\text{sorted}}$ **do**
    Compute shortest path distance $d_H(u, v)$ in H between u and v

    if $d_H(u, v) > t \cdot w_{uv}$ **then**
      Add $e = (v, u)$ to E'

  Return: $H = (V, E', w_{|E'})$

Complexity analysis

The algorithm requires an undirected graph at input, so it is necessary to account for the symmetrization cost, which is $\Theta(m)$. Let us assume the prority queue inside every Dijkstra search is a binary heap, and a stable log-linear edge sorting algorithm.

Note 4 *The sparsification procedure relies on repeated shortest-path distance computations, as edge removal leads to changes in APSPs. Consequently, the overall runtime is directly dependent on the complexity of the chosen APSP algorithm. Given that the input graphs are directed with nonnegative edge weights, this implementation utilizes Dijkstra's algorithm for computing shortest path distances, due to its general applicability and efficiency under these constraints. However, the implementation of this sparsifier is subject to variation and alternative APSP algorithms could be employed depending on specific graph properties.*

Pessimistic running time. During the greedy t-spanner loop, the j-th edge is tested against the current subgraph H_{j-1} containing at most $j-1$ edges.

Single-source Dijkstra with a binary heap can be expressed as

$$T_{\text{Dij}}(j-1,n) = \Theta((j-1)\log n)$$

and the loop cost becomes

$$\begin{aligned} T_{\text{loop}} &= \sum_{j=1}^{m} \Theta(j \log n) \\ &= \Theta\left(\frac{m(m+1)}{2} \log n\right) \\ &= \Theta(m^2 \log n). \end{aligned}$$

The overall worst-case time is $T_W(G) = \Theta(m^2 \log n)$, which for sparse graphs ($m = \Theta(n)$) is $T = \Theta(m^2 \log n)$ and for dense graphs ($m = \Theta(n^2)$) becomes $T = \Theta(m^4 \log n)$.

Expected running time. For these calculations, we adopt the Erdős–Rényi random-graph model denoted as $G \sim G(N,p)$. This means that the vertex set is fixed and has size n, and for every unordered pair of distinct vertices $\{u,v\}$ we independently put the edge (u,v) into E with probability p and leave it out with probability $1-p$ (*edge-inclusion probability*). Nothing about the further analysis will truly depend on a particular numeric value of p; different choices of this value correspond to sparser or denser random graphs.

With this in mind, the random variable $m = |E|$ is therefore binomially distributed:

$$m \sim \text{Bin}(N,p), \quad N = \binom{n}{2}.$$

This means that every one of the N potential edges is included with probability p. For a binomial variable $X \sim \text{Bin}(N,p)$, we have

$$\mathbb{E}[X] = Np, \quad \text{Var}(X) = Np(1-p).$$

Since $\text{Var}(X) = \mathbb{E}[X^2] - (\mathbb{E}[X])^2$, it follows that

$$\mathbb{E}[m^2] = Np(1-p) + (Np)^2.$$

Substituting $N = \binom{n}{2} = \Theta(n^2)$ gives

$$\mathbb{E}[T(m,n)] = \frac{1}{2}\log n \left(Np(1-p) + (Np)^2\right) = \Theta(pn^2 \log n + p^2 n^4 \log n). \tag{8}$$

In summary:

- **Dense graphs** $p = \Theta(1)$ ($m = \Theta(n^2)$): (8) becomes $\Theta(n^4 \log n)$.
- **Sparse graphs** $p = \Theta(1/n)$ ($m = \Theta(n)$): (8) gives $\Theta(n^2 \log n)$.

Best-case remarks.

- If the weight ordering already produces a valid t-Spanner early on, the loop can stop, but the worst-case analysis assumes the pathological situation that every edge triggers a full Dijkstra search.
- Assuming a Fibonacci heap implementation at edge inspection, the Dijkstra search per node would cost $\Theta(|E(H_{j-1})| + n \log n) = \Theta(j + n \log n)$. This would impact the T_{loop} value, producing a complexity of $\Theta(m^2 + mn \log n)$. However, this data structure is not typically present in standard libraries and is rarely used in production code since the constant factors are large and the implementation intricate.
- When edge weights are small integers over a range r, a radix sort takes $\Theta(m + r)$, and if $r = O(m)$, the sorting phase drops to $\Theta(m)$. This does not change the overall bound.
- Using specialized incremental shortest-path data structures can potentially bring the loop down to $\Theta(mn)$ for constant t, but that is outside of the scope of the plain pseudocode of the algorithm discussed in this work.

6.5 MST

Algorithm 5 Minimal Spanning Tree (MST) Sparsifier

```
1:  procedure MST-SPARSIFY(G)
2:      Input: Weighted graph G = (V, E, w)
3:      Output: Spanning tree subgraph H = (V, E', w_|E')
4:
5:      if G is directed then
6:          G_sym ← SymmetrizeGraph(G)
7:          G ← G_sym                          ▷ use symmetrized graph for processing
8:
9:      E' ← Edges of MST from Kruskal's Algorithm on G
10:
11:     Return: H = (V, E', w_|E')
```

Complexity analysis

The `SymmetrizeGraph()` function, which is executed only when G is directed, produces $G_{\text{sym}} = (V, E', w)$. $|E'| \leq 2m$, and so it follows that $m \leq m' \leq 2m$.

The function scans each edge once and inserts at most one reverse copy, therefore

$$T(G) = \sum_{e \in E} \Theta(1) = \Theta(m).$$

The MST is found by Kruskal's algorithm implemented with

- an $\alpha(n)$-amortized disjoint-set structure
- a comparison-based sort of the edge list

and its worst-case cost is $\Theta(m \log n)$, since the sorting dominates.

Pessimistic running time. The Kruskal phase consists of two steps:

1. **Sorting:** assuming a classic MergeSort algorithm, this yields exactly

$$\sum_{i=0}^{\lceil \log 2m-1 \rceil} m = m\lceil log2m \rceil = \Theta(m \log m)$$

   comparisons and moves.

2. **Union-Find loop:** each of the m' edges triggers at most $2 \times \texttt{find} + 1 \times \texttt{union}$, and so

$$T_{\text{U-F}}(G_{\text{sym}}) = \Theta(m\alpha(n)).$$

Because $\alpha(n) \ll \log n$,

$$T_{\text{MST}}(G_{\text{sym}}) = \Theta(m \log m + m\alpha(n)) = \Theta(m \log n),$$

and the combined cost can therefore be expressed as

$$T_W(G) = T_{\text{sym}}(G) + T_{\text{MST}}(G_{\text{sym}}) = \Theta(m) + \Theta(m \log n) = \Theta(m \log n).$$

Expected running time. The algorithm itself is deterministic — randomness is encountered only if the sorting routine is randomized with an unstable sorting algorithm. For comparison-sorting on m keys the expected number of comparisons is still $\Theta(m \log m)$, and all other steps are unaffected by edge weights or order. Consequently

$$\mathbb{E}[T(G)] = \Theta(m \log n),$$

identical to the worst-case bound.

6.6 KOLS

Algorithm 6 KOLS Sparsifier

procedure KOLS-SPARSIFY(G, k, ρ)
  Input: Weighted graph $G = (V, E, w)$, number of BFS runs $k \in \mathbb{N}$, prune rate $\rho \in (0, 1]$
  Output: Sparsified subgraph $H = (V, E', w_{|E'})$

  $f \leftarrow$ map from edge to integer, initialized to 0 ▷ edge frequency counter
  $S \leftarrow$ random sample of k distinct vertices from V ▷ BFS start vertices
  $T_0 \leftarrow []$
    ▷ guarantee spanning tree T_0 using BFS from first seed

  $Q \leftarrow [S[0]]$, visited $\leftarrow \{S[0]\}$

  while Q is not empty **do**
    $u \leftarrow Q$.dequeue()

    for each neighbor v of u **do**
      if $v \notin$ visited **then**
        Add v to Q, visited
        Add edge (u, v) (or $\{u, v\}$ if undirected) to T_0

  for $i = 1$ to k **do** ▷ run k BFS traversals, count edge frequency
    $s \leftarrow S[i]$
    $Q \leftarrow [s]$, visited $\leftarrow []$

    while Q is not empty **do**
      $u \leftarrow Q$.dequeue()

      for each neighbor v of u **do**
        Increment $f(u, v)$ ▷ edge appears in BFS traversal
        if $v \notin$ visited **then**
          Add v to Q, visited

  $E_{\text{other}} \leftarrow E \setminus T_0$
  Sort E_{other} in descending order by $f(u, v)$

  $k_{\text{edges}} \leftarrow \max(|V| - 1, \lfloor \rho \cdot |E| \rfloor)$
  $E' \leftarrow T_0 \cup$ {first $k_{\text{edges}} - |T_0|$ edges from E_{other}}

  Return: $H = (V, E', w_{|E'})$

Complexity analysis

Fix $k \in \mathbb{N}$ as the number of breadth-first searches and a prune rate $\rho \in (0, 1]$. The number of edges kept after pruning is

$$k_{\text{edges}} = \max\{n - 1, \lfloor \rho \cdot m \rfloor\} \leq m.$$

The algorithm consists of the following phases:

1. Initializing the frequency map $f\colon E \to \mathbb{N}$ — $\Theta(m)$
2. Building the guaranteed spanning tree T_0 by one BFS — $\Theta(n+m) = \Theta(m)$
3. Running the remaining $k-1$ traversals; each scans the whole graph in the worst case, which yields
$$\Theta((k-1)(n+m)) = \Theta(km)$$
4. Sorting $E \setminus T_0$ by f (comparison–based) — $\Theta(m \log m)$
5. Copying the first $k_{\text{edges}} - (n-1)$ edges — $\Theta(k_{\text{edges}}) = \Theta(m)$

The term $\Theta(km)$ from step 3 would dominate only when $k \gg \log n$ for dense graphs or $k \gg \log n/n$ for sparse ones, however, k is typically a relatively small constant value compared to the number of vertices. Thus, upon collecting the terms,

$$T(G) = \Theta(m + m \log m). \tag{9}$$

The lower bound $\Omega(m)$ is immediate because the k BFS passes must inspect every edge they traverse.

Pessimistic running time. Equation (9) instantly gives

$$T_W(G) = \begin{cases} \Theta(n^2 \log n), & \text{dense graphs } (m = \Theta(n^2)), \\ \Theta(n \log n), & \text{sparse graphs } (m = \Theta(n)). \end{cases}$$

Expected running time. Model the input as $G(n,p)$, so $\mathbb{E}[m] = \binom{n}{2} p = \Theta(n^2 p)$. Substituting this expectation into (9) yields

$$\mathbb{E}[T(G)] = \Theta(n^2 p + n^2 p \log(n^2 p) + n). \tag{10}$$

- **Dense graphs** $p = \Theta(1)$ $(m = \Theta(n^2))$: equation (10) produces $\Theta(n^2 + n^2 \log n) = \Theta(n^2 \log n)$.
- **Sparse graphs** $p = \Theta(1/n)$ $(m = \Theta(n))$: equation (10) gives $\Theta(n^2 + n^2 \log n) = \Theta(n \log n)$.

These bounds are tight up to the usual hidden constants whenever k is at least a fixed positive integer.

7 Experiments

7.1 Experimental setup

Graph preprocessing

Due to limitations in access to high-performance computing resources capable of handling real-world large-scale graphs with millions of edges and vertices, series of random graphs from various families were generated using a script, which enabled the execution of experiments across a diverse set of graph topologies while ensuring computational feasibility. A total of 100 distinct random graphs, 25 per graph family, were generated to facilitate the systematic investigation of graph-theoretic properties and the

performance of various algorithms, all within the constraints of available computational resources. This of course leaves a natural area for exploration in any future work regarding these experiments, where the scalability and performance of these algorithms could be further studied on more complex and large-scale networks.

The experiments were conducted on graphs that adhered to the following assumptions:

1. The input graphs were assumed to be devoid of isolated vertices, as such vertices do not contribute meaningfully to the evaluation of graph metrics.
2. For any initially disconnected graph, the largest connected component was extracted, and all subsequent experiments were performed on this component. This procedure ensures that every graph under consideration is connected, thereby making it suitable for the application of the selected metrics and enabling a comprehensive comparative analysis of the sparsifiers' performance. Furthermore, this approach allows for the assessment of whether a given reduction algorithm leads to an actual *loss* of connectivity in comparison to the original graph.
3. Any directed graph underwent the previously discussed symmetrization procedure to accommodate sparsifiers that do not operate on directed graphs. For other sparsifiers, the original directed version of the graph was preserved.

Software framework

The software environment for all experiments was developed using Python 3.9 and integrated a range of custom and open-source modules. Graph data structures and algorithms were handled primarily using the `networkx` library, which facilitated graph manipulation, metric computation, and shortest path analysis. For efficient numerical operations and matrix-based calculations, the `numpy` and `scipy` libraries were employed.

The core graph sparsification algorithms were implemented from scratch in modular form to allow for consistent benchmarking across varying conditions and metrics. General and distance metrics were computed using a combination of native Python routines and custom utilities to ensure flexibility and mathematical transparency in result verification.

The source code, data, and other artifacts have been made available at `https://github.com/kolaowalska/graph-sparsification`.

Hardware platform

All computational experiments were conducted on a personal machine equipped with an Intel Core i9-14900HX CPU and 32 GB DDR5 RAM. Given the memory constraints, synthetic graphs were generated with a maximum edge count of 1,000 to guarantee tractable runtimes and avoid performance bottlenecks. No GPU acceleration was used, and all code was executed in a single-threaded environment under a Linux-based operating system.

8 Results

In this chapter, we evaluate how previously discussed graph sparsification algorithms (i) reduce the number of edges, (ii) affect the running time of metric computation on sparsified graphs, and (iii) how the observed behavior lines up with the previously calculated theoretical complexities.

The sparsification methods presented in this study can be evaluated along many axes, including structural, spectral, and distance-based metrics, both within individual graph families and in cross-family comparisons. While a complete analysis could involve an exhaustive breakdown across all such dimensions, we limit our focus to a representative subset of the most informative metrics in order to highlight the core differences between algorithms and draw meaningful conclusions while maintaining clarity and conciseness.

8.1 Edge reduction strength

Figure 5 presents a quantitative analysis of edge reduction facilitated by sparsifiers across different graph families.

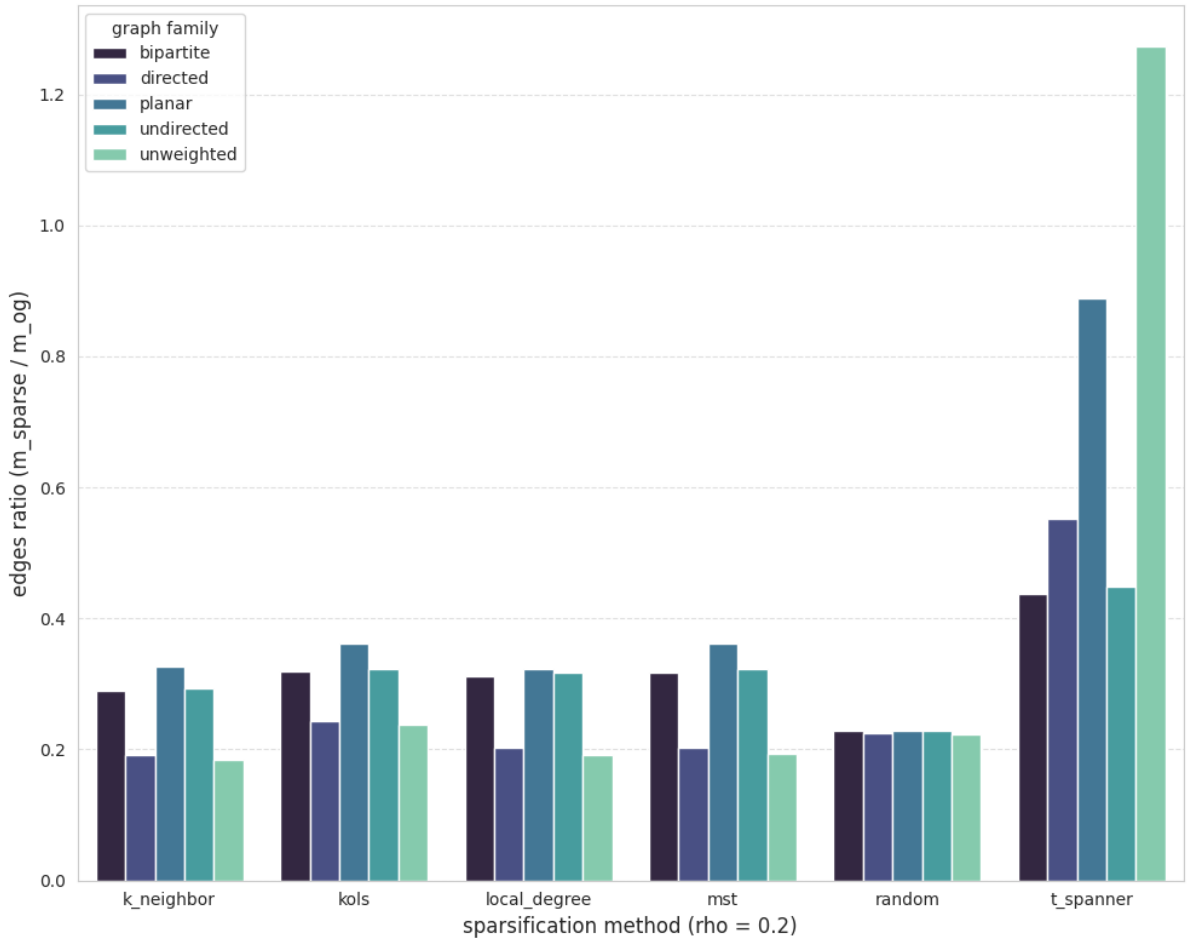


Figure 5: Edge ratio for each family upon sparsification.

K-Neighbor, KOLS, Local Degree and *MST* are the most aggressive pruners, driving the edge set down to roughly $20 - 35\%$ of its original size in every family. The small inter-family spread suggests that their

heuristics are not overly sensitive to topological peculiarities. *Random* sampling removes a comparable fraction of edges, but its variance across families is larger since high-degree hubs are discarded with the same probability as low-degree vertices. *t-Spanner* is an outlier: for planar and undirected graphs it keeps $\sim$80% of the edges, while for unweighted graphs it *adds* edges ($m_{\text{sparse}}/m_{\text{og}} \approx 1.3$). This behavior is consistent with the algorithm's guarantee to preserve stretch at the cost of extra edges when the input graph is already sparse.

8.2 Metric computation time

Figure 6 shows the wall-clock time needed to compute all metrics on sparsified graphs. Figure 7 shows that time normalized by the metric computation time on the original graph (negative values indicate a speed-up).

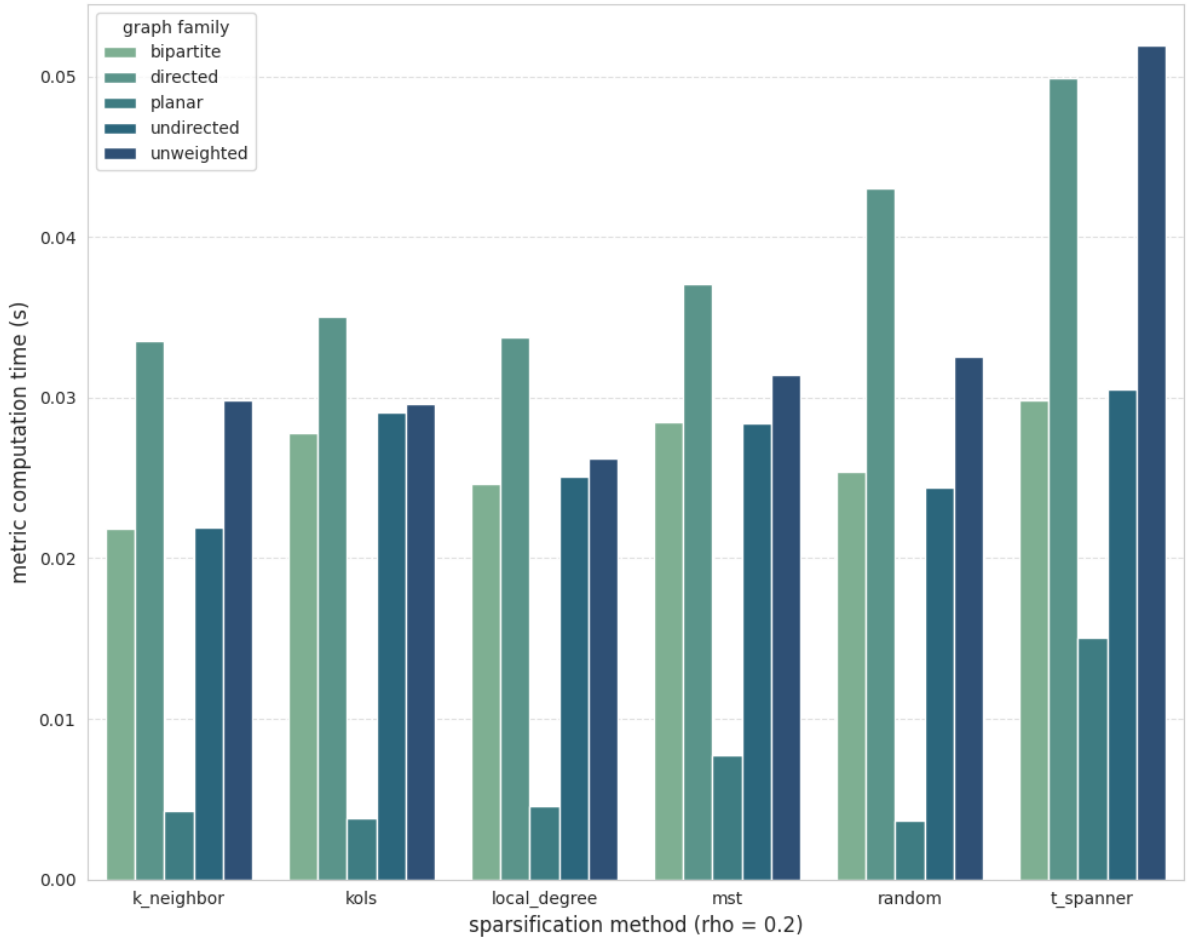


Figure 6: Average metric computation time on sparsified graphs across different graph families.

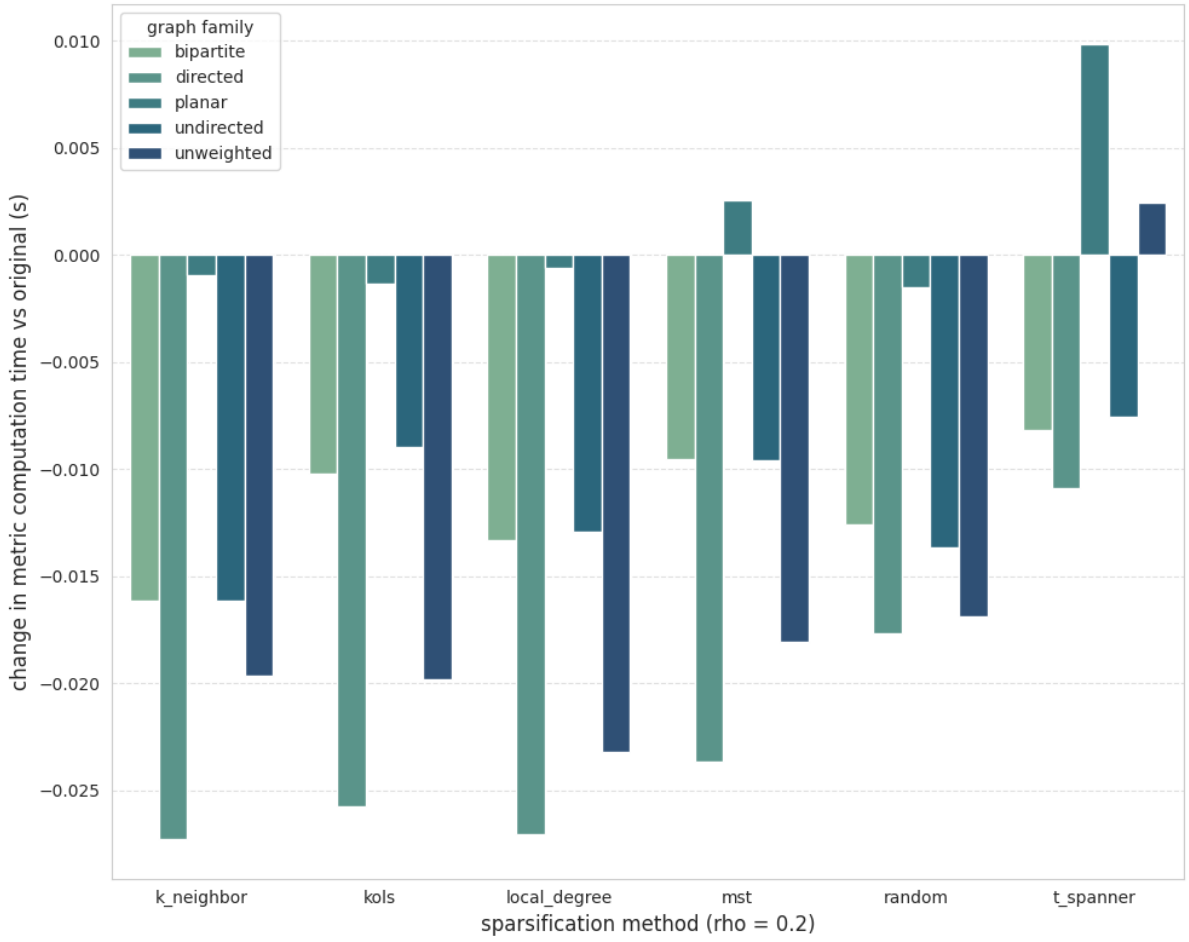


Figure 7: Change in average metric time computation upon sparsification across graph families.

The four strongest pruners *K-Neighbor, KOLS, Local Degree* and *MST* translate their lower edge counts into an appropriate reduction in metric time across every graph family. *Random* offers only a modest speed-up, reflecting the fact that purely random edge reduction may not eliminate the densest parts of the graph and does not always preserve key connections, leading to a loss of shorter paths between vertices. *t-Spanner* is the only method that can be slower than using the original graph: for planar and unweighted graphs the metric phase takes visibly longer. This matches the earlier observation that it sometimes increases the edge count.

8.3 Relation to theoretical complexity

One-shot use. When each graph is sparsified once and then analyzed, the dominant cost is typically the metric computation phase, which is roughly linear in regard to the number of edges. Hence the empirical ordering of speed-ups closely follows the edge-reduction strength, not the worst-case complexity of the sparsifier itself. For example, the *Local Degree* sparsifier has a higher theoretical construction cost

($O(|V|\log|V|)$) than Random ($O(|E|)$), yet it delivers a larger net time saving because it prunes more aggressively.

Repeated queries. If the same graph is analysed many times, the up-front cost of building the sparsifier becomes a nontrivial contribution to the overall runtime. Here the theoretical bounds give useful guidance:

Algorithm	**Construction complexity**	**Practical Implication**
Random	$O(\|E\|)$	Fastest to build; good for very large graphs with few queries.
Local Degree	$O(\|V\|\log\|V\|)$	Scales well; pay once, save on every metric call.
KOLS, K-Neighbor	$O(\|V\|^2\log\|V\|)$ or $O(\|V\|^2)$	Viable on graphs with $\|V\| \lesssim 10^4$; otherwise, construction dominates.
MST	$O(\|E\|\log\|V\|)$	Competitive provided the input is not extremely dense.
t-Spanner	$O(\|E\|^4\log\|V\|)$	Potentially impractical beyond toy instances.

8.4 Sparsification time

In Figure 8 the sparsification phase itself exhibits clearly separated time–scales that line up with the theoretical construction costs.

- **Minimum tier.** *Random* sampling is consistently the fastest builder, completing in a fraction of a millisecond for every graph family. This matches its linear $O(|E|)$ complexity and the fact that it touches each edge only once, without any global bookkeeping.
- **Low tier.** *Local Degree* and *MST* form the next cluster. Both need a lightweight priority queue or sort, giving the expected $O(|V|\log|V|)$ and $O(|E|\log|V|)$ behavior. Their finishing times are almost an order of magnitude below the quadratic algorithms, yet an order of magnitude above pure random sampling.
- **Middle tier.** *K-Neighbor* and *KOLS* incur a noticeably higher construction cost. Because they perform work proportional to $|V|^2$ (with an additional $\log|V|$ factor for *KOLS*), their wall-clock times scale in a closely linear relation with graph size. In random benchmarks they are roughly two to three times slower than *Local Degree*, irrespective of graph family.
- **Outlier.** The t-Spanner implementation is one to two orders of magnitude slower than every other method — even on modest graphs — due to its $O(|E|^4\log|V|)$ worst-case behavior. This makes it practical only for very small inputs or scenarios where low stretch is absolutely critical and preprocessing time is irrelevant.

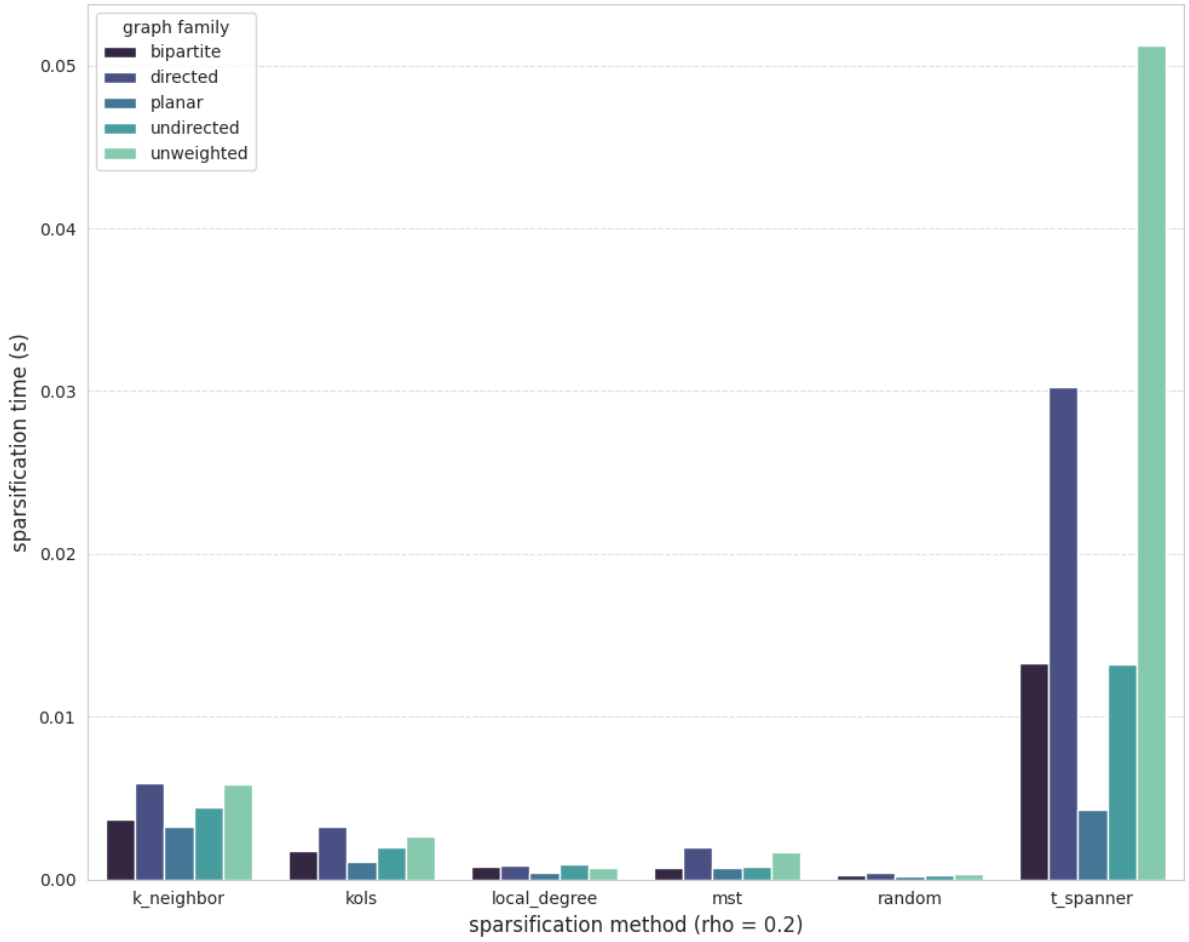


Figure 8: Average sparsification time across graph families.

Family dependence. Although each graph family has its own constant factors (planar graphs tend to sparsify fastest, unweighted graphs slowest), we can observe that *the relative ordering of the algorithms never changes*. This reliability is highly advantageous in practice: one can choose a sparsifier based on the asymptotic guidance alone and expect the same hierarchy of construction times on real-world data.

Practical takeaway. If preprocessing time is the primary constraint, *Random* sampling is a clear leader. If a moderate one-off cost is acceptable in exchange for a stronger subsequent speed-up of graph analytics, *Local Degree* or *MST* provide the best balance. Quadratic methods (*K-Neighbor*, *KOLS*) are suitable for graphs with a few thousand vertices, while the t-Spanner can be reserved for niche, stretch-sensitive applications.

8.5 Summary of Results and Insights

The aim of this work was to present a systematic comparative analysis and evaluation of multiple graph sparsification methods aimed at optimizing shortest path calculation. The performance of each method was assessed based on the effectiveness in preserving essential graph metrics. Key findings demonstrate that no single algorithm universally outperforms others across all tested metrics and graph families. The computational complexity analyses provided theoretical justifications for observed performance differences, highlighting practical considerations for algorithm selection in real-world applications. The purpose of these insights is to to facilitate more informed decisions regarding the integration of reduction algorithms into computational workflows, maintaining a balance between computational efficiency and result accuracy. This study means to emphasize the importance of context-dependent selection of sparsification algorithms, underscoring the necessity for a suited approach based on application-specific requirements and dataset characteristics.

Potential future directions

While this particular work addressed the rudimentary aspects of graph sparsification in shortest path optimization, several avenues remain open for future exploration of this topic.

- **Scalability Analysis:** Extending experimental setups to accommodate large-scale graphs with millions of vertices and edges, providing valuable insights into the scalability and practical limitations of these algorithms.
- **Algorithmic Enhancement:** Developing hybrid sparsification algorithms that combine strengths of individual methods, potentially yielding superior preservation of graph metrics.
- **Dynamic Graph Sparsification:** Exploring reduction algorithms adapted specifically for dynamically changing graphs and assessing the feasibility and effectiveness of incremental sparsification techniques.
- **Real-world Case Studies:** Conducting detailed studies on real-world network datasets, such as communication networks or transportation systems, to validate theoretical insights and further refine algorithmic approaches.
- **Machine Learning Integration:** Investigating the incorporation of machine learning techniques to dynamically select or optimize reduction strategies based on graph topology and intended computational tasks.

References

[1] I. Althöfer, G. Das, D. Dobkin, and D. Joseph. Generating sparse spanners for weighted graphs. In *SWAT 90: Proc. of the 2nd Scandinavian Workshop on Algorithm Theory*, volume 447, pages 26–37. Springer, 1990.

[2] Partha Basuchowdhuri, Satyaki Sikdar, Sonu Shreshtha, and Subhashis Majumder. Detecting community structures in social networks by graph sparsification. In *Proceedings of the 3rd IKDD Conference on Data Science, 2016*, pages 1–9, 2016.

[3] Y. Chen, H. Ye, S. Vedula, A. Bronstein, R. Dreslinski, T. Mudge, and N. Talati. Demystifying graph sparsification algorithms in graph properties preservation. *arXiv preprint arXiv:2311.12314*, 2023.

[4] M. Hamann, G. Lindner, H. Meyerhenke, C. L. Staudt, and D. Wagner. Structure-preserving sparsification methods for social networks. *arXiv preprint arXiv:1601.00286*, 2016.

[5] J. McAuley and J. Leskovec. Learning to discover social circles in ego networks. In *Advances in Neural Information Processing Systems (NIPS)*, volume 25, 2012.

[6] Christian L. Staudt, Aleksejs Sazonovs, and Henning Meyerhenke. Networkit: A tool suite for large-scale complex network analysis, 2014.