Neighborhood-Preserving Graph Sparsification

Abd Errahmane Kiouche, Julien Baste, Mohammed Haddad, Hamida Seba, Angela Bonifati

Neighborhood-Preserving Graph Sparsification*

Abd Errahmane KIOUCHE[1] Julien BASTE[2] Mohammed HADDAD[1]
Hamida SEBA[1] Angela BONIFATI[1]

[1] Universite Claude Bernard Lyon 1, CNRS,
INSA Lyon, LIRIS, UMR5205, 69622 Villeurbanne, France
[2] Univ. Lille, CNRS, Centrale Lille,
UMR 9189 - CRIStAL F-59000, Lille, France

abderrahmane.kiouche@gmail.com, julien.baste@univ-lille.fr, mohammed.haddad@univ-lyon1.fr,
hamida.seba@univ-lyon1.fr, angela.bonifati@univ-lyon1.fr

Abstract

We introduce a new graph sparsification method that targets the neighborhood information available for each node. Our approach is motivated by the fact that neighborhood information is used by several mining and learning tasks on graphs as well as reachability queries. The result of our sparsification technique is a sparsified graph that can be used instead of the original graph in the above tasks while still ensuring fairly good approximations for the results. Moreover, our sparsification method allows users to control the size of the resulting sparsified graph by adjusting the amount of information loss tolerated by the targeted applications. Our extensive experiments conducted on various real and synthetic graphs show that our sparsification considerably reduces the size of the graphs by achieving 40% sparsification rate on average on several input graphs. Furthermore, in the experimental study we show the utility and efficiency of our sparsification algorithm for notable data-driven tasks, such as node classification, graph classification and shortest path approximations. The obtained results exhibit interesting trade-offs between the runtime speed-up and the precision loss.

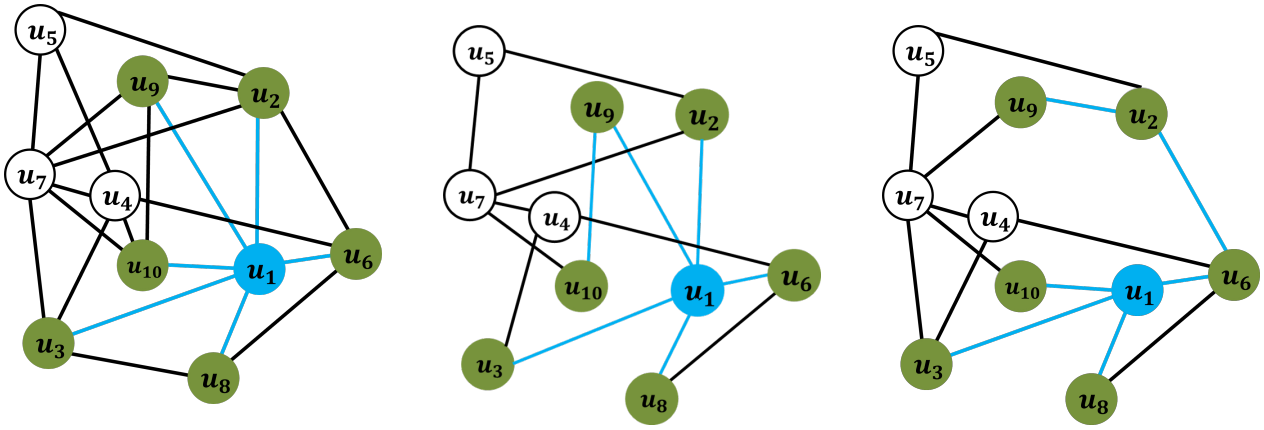
1 Introduction

Graphs are data modeling abstractions consisting of a set of vertices, also called nodes, and a set of edges connecting the vertices. Vertices represent objects, while edges represent relationships between them. Graphs are widely used in data modeling because of their ability to represent, in a simple and intuitive way, complex processes in both nature and technology, such as social interactions, protein-protein interactions, chemical molecules, transport networks and fraud detection networks, to name a few [26]. However, most of these graphs are very large or grow exponentially as new data arrives. This makes graph querying and analysis a very challenging task.

To tackle scalability and performance issues when dealing with large graph data, plenty of algorithms are devised to simplify graphs in several domains and applications related to graph analysis [19]. The aim is to construct simpler or smaller representations for large graphs mainly to save storage space but also to use the obtained representations, instead of the original graphs, in applications where using the entire large original graphs is not possible or is time consuming [16]. Sparsification is one of these approaches aiming to construct a subgraph of the original graph by removing insignificant edges. The resulting graph is called a skeleton or a backbone. Sparsification is generally application-dependent because the significance of an edge may vary from one application to another. The main idea is obtaining a smaller graph while preserving some properties, even approximately, of the original graph such as the results of distance, and reachability queries [9, 18].

Several sparsification methods are proposed in the literature [20] but there is no generic approach that can target several applications at once. An attempt to build a fairly general approach is provided in [32], relying on reinforcement learning. However, this method does not exempt us from computing the application task on the original graph; on the contrary, this step is mandatory during the training process that needs to

*This is a preprint: the final reviewed paper will appear in the proceedings of the 51st International Conference on Very Large Data Bases (https://vldb.org/2025/)



(a) Input Graph (b) $p(1) = 50\%$ $t = 2$ (c) $p(1) = 50\%$ $t = 3$

Figure 1: $(p, t)-$sparsification and neighborhood information.

be achieved on each graph to be sparsified. This makes it less useful for real-world applications and hard to use for graphs that the algorithm has not used during training.

In this paper, we fill the gap by proposing a general sparsification method, called $(p, t)-$Sparsification, that focuses on preserving the neighborhood information available around each vertex. This kind of local information is a key property for many graph algorithms but it has never been exploited for the graph sparsification problem. For instance, in graph learning tasks such as node classification, knowing a node's immediate connections is crucial for accurately categorizing it based on the characteristics of its neighbors. Similarly, for reachability queries, where the goal is to find whether there is a path between two nodes, neighborhood information speeds up the process and makes it more efficient. Computing shortest paths in a graph needs also the information about each node's local surroundings to identify the most efficient and least costly paths.

Each node in a graph may have several connections, and handling all of them is time and space consuming. Hence, our approach allows to finely control how much of locality we keep in the obtained sparified graph. The key idea is to remove some edges while ensuring that for each node u, a certain amount of its neighbors, determined by a function $p()$, is kept within t-hops at most from u in the resulting sparsified graph, with $t \geq 1$ and $p()$ being the input parameters.

Figure 1 shows how the neighborhood information is preserved with (p, t)-sparsification. In the original graph (cf. Figure 1 (a)), node u_1 has 6 direct neighbors (highlighter in blue): nodes u_2, u_3, u_6, u_8, u_9 and u_{10}. Figure 1 (b) shows a $(p, t)-$sparsification of G where $p(1) = 50\%$ of the neighbors of node u_1 remain reachable at $t = 2$ hops. We can observe that node u_2 is no longer directly connected to node u_1 in the sparsified graph but remains reachable within 2 hops from it. Figure 1 (c) shows another sparsification of G this time with $p(3) = 100\%$ of the neighbors of node u_1 remaining reachable within $t = 3$ hops. We can also see that nodes u_2 and u_9 are no longer directly connected to node u_1 in the sparsified graph but remain reachable within 3 hops from it. By varying the (p, t) parameters, specified as input, one can see that the obtained sparsification can be accordingly tuned, thus explaining its generalizability to any input graph.

The example of Figure 1 also shows that parameters p and t allow to have a generic approach that can produce various sparsifications by using different values for p and t and therefore adapt to multiple applications and needs. To show this, we use our resulting sparsified graphs in four main tasks: reachability queries, shortest paths computation, node classification and graph classification. The results we obtained on these four different tasks have proven to be effective approximations of the exact results of these tasks obtained on the the full original graphs. This validates our method as producing useful sparsified graphs reducing the size of the input graph, and allowing to leverage the sparsified graphs, instead of the original ones, without sacrificing the accuracy of the outcomes. This efficient approximation is crucial in computational environments where resource savings are imperative, yet the integrity of the results must be maintained. It is also interesting in several applications where a quickly delivered approximate result is more useful than an accurate result that takes a long time to obtain.

Summarizing, the main contributions of this paper are as follows:

- We introduce a new general graph sparsification method that targets the preservation of the neighborhood information available for each node in the graph. The key idea is to create a graph skeleton that can replace the original graph in various applications and for arbitrary graph datasets.

- Our method allows the fine-tuning of the size of the resulting sparsified graph by adjusting the amount of preserved neighborhood information.
- Our method allows efficient graph algorithm approximations. Our approach improves the efficiency of various graph algorithms, particularly those that depend on node neighborhood information, such as node/graph classification and shortest path computation. This results in faster and more robust approximations for larger graphs.
- Our techniques allows to strike a user-driven balance between sparsification ratio and information loss through its parameters p and t. The user-driven customization makes our technique versatile for different applications with varying needs.
- We conduct extensive experiments using real-world datasets in different application domains to assess the effectiveness and efficiency of the proposed method. The source code and the data used in our paper are publicly available at https://gitlab.liris.cnrs.fr/coregraphie/ptspar.

The remainder of this paper is organized as follows: Section 2 defines the basic concepts and notation used in the paper and reviews related work on graph sparsification methods. Section 3 formally defines the problem of neighborhood-preserving graph sparsification and studies its complexity. Then, Section 4 provides a description of the algorithms that we propose to compute this sparsification and analyses their time complexity. Section 5 presents the results obtained through the extensive experiments we undertook to evaluate our sparsification approach, as well as the usefulness of the obtained sparsified graphs. Finally, Section 6 concludes the paper and points out some research perspectives.

2 PRELIMINARY AND RELATED WORK

2.1 Preliminary

A graph $G = (V, E)$ is a 2-component structure comprising a set V of vertices and a set $E \subseteq V \times V$ of edges connecting the vertices. Edges can be directed, and both vertices and edges can have attributes or weights. In this paper, we consider unweighted and undirected graphs. So, an edge connecting vertices u and v is interchangeably denoted by uv or vu.

Two vertices connected by an edge are said to be adjacent. A vertex adjacent to v is called a direct neighbor of v or a 1-hop neighbor of v. The set of all the direct neighbors of a vertex v in G is denoted as $N^1_G(v)$ or simply $N(v)$ when there is no ambiguity. The degree of a vertex v, denoted $deg(v)$, is the number of its 1-hop neighbors, i.e., $deg(v) = |N(v)|$.

A path in a graph is a sequence of edges which joins a sequence of vertices which are all distinct. A vertex v is reachable from a vertex u if there exists a path from u to v. A *q-hop neighbor* of a vertex v is a vertex that can be reached from v with a path of exactly q edges. We will denote by $N^h_G(v)$ the set of all $(1 \leq q \leq h)$-hop neighbors of vertex v in G.

The distance between two vertices u and v is the length, i.e., number of edges, of the shortest path connecting them. We will denote by $\mathcal{W}$ the set of all paths in G, by $\mathcal{W}uv$ the set of all paths from node u to node v, and by $\mathcal{W}uv^i$, the set of paths from node u to node v of length at most i.

Graph sparsification stands for the methods that compute a sparse subgraph of the input graph. Given a graph $G = (V, E)$, a sparsified graph of G is a graph $G_s = (V_s, E_s)$ defined generally on the same set of vertices as G but with less edges, i.e., $V_s = V$ and $E_s \subset E$. In practice, we want the sparsified graph G_s to retain certain properties of G such as the distance between vertices, reachability queries, etc. A graph sparsifier that preserves distances between nodes is generally called a k-spanner.

When sparsifying a graph, the order in which the edges are processed is generally important. We will denote this order with a bijective function, i.e., permutation, $\pi : E \to \{1, \ldots, |E|\}$ that associates to each edge $e \in E$ its processing rank $\pi(e)$. We will denote by E^π the edges of G in the order defined by π.

Let $G(V, E)$ be the input graph and $G_s(V_s, E_s)$ be the sparsified graph, the sparsification ratio measures how well G_s reduces the graph G and is given by the ratio of the number of deleted edges over the total number of edges:

$$Sr = \frac{|E| - |E_s|}{|E|} \tag{1}$$

Note that the higher is the sparsification ratio the better is the storage space gain ensured by the sparsification.

Table 1 summarizes these notations.

Table 1: Notation.

Symbol	Description
$G(V, E)$	an undirected unweighted graph with V the set of vertices and E the set of edges.
$N_G^h(v)$	the set of all $(1 \leq i \leq h)$-hop neighbors of vertex v in G
$N(v)$	the set of direct neighbors of v, i.e., $N_G^1(v)$
$deg(v)$	number of direct neighbors of v
d	average node degree of G
$G_s = (V_s, E_s)$	a sparsified graph of G
π	an ordering function defined on E
E^π	the edges of G in the order defined by π
$\mathcal{W}$	the set of all paths in G
$\mathcal{W}_{uv}$	the set of all paths in $\mathcal{W}$ from u to v
$\mathcal{W}_{uv}^i$	the set of all paths of $\mathcal{W}_{uv}$ of length at most i that pass through the edge e
Sr	the sparsification ratio given by $\frac{\lvert E\rvert - \lvert E_s\rvert}{\lvert E\rvert}$

2.2 Related Work

The first graph sparsifiers are graph spanners introduced in [24] and motivated by the problem of distance similarity of two graphs. Graph sparsification is also used in [1] to speed-up algorithms computing graph cuts, which are partitions of the vertices of a graph into two disjoint subsets. Given any weighted undirected graph $G = (V, E)$, the authors show that one could construct a new graph $G_\varepsilon = (V, E_\varepsilon \subseteq E)$, $0 < \varepsilon < 1$, with $|E_\varepsilon| = O(n \log n/\varepsilon^2)$ edges such that the value of every cut in G is within a multiplicative factor of $1 \pm \varepsilon$ of its value in G_ε.

Nowadays, several other properties are preserved through graph sparsification such as spectral similarity of graph Laplacians [29], determinant-preservation of matrices [8], to reduce the number of constraints in the binary constraint satisfaction problem (CSP)[4], etc.

Given a social graph and a log of past traversals of this graph, the authors of [21] prune the graph to a prefixed extent, while maximizing the likelihood of generating the traversal traces in the log. A similar work is described in [2]. It tackles the problem of simplifying a graph, while maintaining the connectivity recorded in a given set of observed activity traces represented by a set of trees with specified roots. The problem consists in selecting a subset of edges in the graph so as to maximize the number of nodes reachable in all trees by the corresponding tree roots. Many works have also been proposed to extract the backbone of graphs by removing edges based on various properties of graph vertices and edges such as degree distribution or betweenness centrality distribution [14, 34]. These methods rely mainly on edge weights to sparsify the graph. So, they perform poorly on unweighted graphs. As examples, we can cite statistical methods that apply a filter on the edges such as the Noise Corrected filter [6] that keeps only the edges that have a weight greater than a given threshold. High salience backbone filter [9] extracts the graph skeleton based on the link (i.e., edge) salience property. The salience of an edge e is a score $s(e)$ that represents a consensus estimate from all nodes of the importance of the edge e. An edge e having a salience score equal to 1.0 is an essential edge for all nodes. If $s(e) = 0$, the edge e has no role and if, $s(e) = 0.5$ then it is important for only half of the nodes [9]. The salient backbone extracts the skeleton by keeping only the edges that have a salience score greater than a certain threshold.

More recently, Wickman et al. [32] propose SparRL a graph sparsification approach based on graph neural networks (GNNs) and reinforcement learning. However, SparRL requires executing the downstream task algorithm on the original graph to calculate rewards for the reinforcement learning part. This raises questions about the practical utility of this sparsification method and limits its generalization to unseen graphs. Also, the method applies a task-specific optimization which is difficult to apply to some task such as learning and classification.

Graph sparsification methods are generally designed for specific applications because it is difficult to have sparsified graphs that can be used in several kind of graph applications. By targeting neighborhood information and allowing to control the amount of information loss in the computed sparsified graph, we aim to be able to use our skeletons in a variety of graph applications. In fact, several graph algorithms, such as node embedding, node classification, shortest paths, etc. are based on the availability of node neighborhood information. In the remainder of the paper, we show that controlling the amount of this information in the computed skeleton allows to reach good trade-off between algorithm speed-up and precision loss when using

the skeleton as input instead of the original graph in the targeted applications.

3 A Neighborhood-preserving graph sparsification

In this section, we introduce a new graph sparsification method that targets the amount of neighborhood information available for each node in the graph. The main idea is to sparsify the input graph by removing edges, while ensuring that, for all $1 \leq i \leq t$, a proportion $p(i)$ of the neighbors of each node v is included in the set of the i-hops neighbors of v in the resulting sparsified graph, where $t \geq 1$. We denote such sparsification by (p,t)-sparsification where:

- $p : N^* \rightarrow [0,1]$ is a monotonically increasing function, which represents the proportion of each node's input neighbors that must be available in its i-hops neighborhood in the sparsified output graph.
- t : is the minimum integer value for which p reaches its maximal value *i.e.*, $p(i) = p(t), \forall i \geq t$.

More formally, given an undirected graph $G = (V, E)$, a (p,t)-sparsification of G is defined as follows:

Definition 1 *Given a positive integer t and a monotonically increasing function $p : N^* \rightarrow [0,1]$ satisfying $p(i) = p(t)$ for all $i > t$, a* (p,t)-sparsification *of a graph $G = (V, E)$ involves finding a subgraph $G_s = (V_s, E_s)$ of G. G_s must have the same set of vertices $V_s = V$, a subset of edges $E_s \subseteq E$, and must satisfy the condition that for each integer $0 < i \leq t$ and each vertex $v \in V$, the set $N^i_{G_s}(v)$ of includes at least a proportion $p(i)$ of the set $N^1_G(v)$ of immediate neighbors of v in G.*

The definition implies that the subgraph G_s retains fewer edges than the original graph G, but still captures a specified proportion of the original neighborhood structure.

With (p,t)-sparsification, the function p aims to control the loss of neighborhood information at varying depths. Naturally, a smaller value of p results in a higher sparsification ratio and vice versa.

For any (p,t)-sparsification, the number of edges $|E_s|$ of the sparsified graph satisfies the inequality $|E|p(1) \leq |E_s|$.

The proof is straightforward and follows from the handshaking lemma which states that in any graph, the sum of the degrees of all the vertices is twice the number of edges.

Theorem 1 *Finding the optimal (smallest) graph satisfying the (p,t)-sparsification constraints for $t \geq 2$ is an NP-Hard problem.*

[1] The proof of the theorem follows directly from hardness of finding k-spanners which is known to be NP-complete [25].

4 Computing (p,t)-sparsifiers

In this section, we present two main algorithms for finding (p,t)-sparsifiers of an input graph G. The first algorithm is an exact algorithm based on an integer linear programming (ILP) formulation of (p,t)-sparsification. The second algorithm is an approximation whose result depends on the order on which the edges are processed, thus we provide several solutions to this ordering problem.

4.1 Exact Algorithm

Our exact algorithm is obtained by solving an Integer Linear Programming (ILP) formulation of (p,t)-sparsification. This formulation is aimed at finding an optimal (smallest) sparsified graph that meets the (p,t)-sparsification definition. It consists of a set of linear inequalities that constrains the minimization of an objective function. In our case, the objective function counts the number of edges of the sparsified graph (cf. Equation 4.1) and the constraints are expressed by Inequalities 4.1 to 4.1 and domain definition of our variables (cf. Equation 4.1).

Given a graph $G = (E, V)$ and a (p,t)-sparsification, the following ILP formulation computes a smallest sparsified graph as follows:

minimize $\sum_{e \in E} x_e$

subjectto

$\sum_{v \in N(u)} \sum_{w \in \mathcal{W}^i_{uv}} x_w \geq p(i) \cdot |N(u)| \forall u \in V, i \leq t$

[1]The detailed proofs are provided in the supplementary material also available at https://gitlab.liris.cnrs.fr/coregraphie/ptspar/-/blob/main/Supplementary$_m$aterial.pdf

$\sum_{w \in \mathcal{W}_{uv}} x_w \leq 1 \forall uv \in E$
$x_w \leq x_e \forall w \in \mathcal{W}, e \in w$
$x_e, x_w \in \{0,1\} e \in E, w \in \mathcal{W}$
where a binary variable x_e is defined for every edge $e \in E$ such that $x_e = 1$ if and only if the edge e is selected to be part of the sparsified graph, otherwise $x_e = 0$. Thus, the objective function $\sum_{e \in E} x_e$ aims at minimizing the number of selected edges in the final solution. The first constraint (cf. Equation 4.1) ensures that for every vertex u of the graph, the property of (p,t)-sparsification is satisfied *i.e.,* for every distance $i \leq t$, the number of neighbors still connected to u via a path of length at most i is at least $p(i) \cdot |N(u)|$. This is enforced by the binary variables x_w such that for every neighbor v of u, the set of all paths w between u and v is denoted by $\mathcal{W}_{uv}$ and by $\mathcal{W}^i_{uv}$ when considering paths of length i, hence $\sum_{v \in N(u)} \sum_{w \in \mathcal{W}^i_{uv}} x_w \geq p(i) \cdot |N(u)| \forall u \in V, i \leq t$. The second constraint (cf. Equation 4.1) makes sure that the obtained sparsified graph has no cycles *i.e.,* there is at most one path between any pair of vertices. This is enforced by setting the x_w to 1 to at most one path w among all possible paths $\mathcal{W}_{uv}$ between two adjacent vertices u and v in G. The constraint given by Equation 4.1 ensures that all edges belonging to a selected path w are selected in the sparsified graph. The last constraint (cf. Equation 4.1) sets the definition domain of x_e, x_w variables which are defined as binary variables.

4.2 Approximation Algorithm: ptSpar

We propose *ptSpar* (see Algorithm 1) an approximation algorithm that implements (p,t)-sparsification. It takes as input a graph $G = (V, E)$ to sparsify, the sparsification parameters p and t and an ordering E^π for processing the edges of the input graph.

The algorithm starts with an empty sparsified graph G_s (see line 1) and grows it incrementally by going through all the edges of the input graph G, in the order E^π (see the loop on lines 3 to 18). G' is a working variable initialized to the empty graph and serves to check that an edge inserted in G_s verifies the neighborhood conditions of the (p,t)- sparsification. Each iteration of the loop (lines 3 to 18) corresponds to the processing of a new edge e in the ordering E^π. A processed edge e, is first inserted into G' (line 4) but its inclusion in the sparsified graph G_s depends on whether it verifies the condition of (p,t)-sparsification. This is done by setting variable *insert* to false (line 5).

To see whether edge $e = uv$ needs to be included in G_s, we simply check if G_s without the edge e remains a (p,t)-sparsification for G'. To do so, we check the neighborhood preservation constraints for nodes u and v as they are the only nodes whose neighborhood set is impacted by the arrival of edge e. For such purpose, we need to compute the set of all neighbors of u and v located in a radius <= i, i.e., $N^i_{G_s}(u)$ and $N^i_{G_s}(v)$ (see lines 9-10). To compute $N^i_{G_s}(u)$, respectively $N^i_{G_s}(v)$, we traverse the graph within radius i starting from u, respectively v. Then, we check if the non-insertion of e in G_s violates the neighborhood preservation constraints (line 11), if this is the case e must be inserted in G_s (lines 12-16).

Theorem 2 *The subgraph $G_s = (V_s, E_s)$ output of Algorithm ptSpar is a (p,t)-sparsification, of the input graph $G = (V, E)$.*

We proceed by induction and show that if $G_s(k)$ is a (p,t)-sparsification of $G'(k)$, then $G_s(k+1)$ must also be a (p,t)-sparsification of $G'(k+1)$, where k denotes the iteration step in the algorithm, representing the stage at which the edges are processed. This is demonstrated by showing that an assumption of the contrary leads to a contradiction with the induction hypothesis, thereby confirming the theorem's claim through induction.

The performance of the *ptSpar* algorithm is significantly impacted by the order in which edges are processed. Different edge orderings can lead to varying efficiencies in achieving (p,t)-sparsification. Optimizing the (p,t)-sparsification for a graph G fundamentally involves finding the most effective edge ordering, $E^{\pi *}$, of the edges as stated in Theorem 3.

Theorem 3 *Let $G = (V, E)$ be a graph. There exists a permutation function π^* of the edge set E for which algorithm ptSpar, gives an optimal (i.e., a minimum size (p,t)-sparsification) of G.*

The proof begins by asserting that if at any iteration k, the current output $G_s(k)$ is a (p,t)-sparsification of G, subsequent edges processed will be rejected, maintaining $G_s(k)$ unchanged until the final iteration. This is based on the observation that once a graph meets the (p,t)-sparsification criteria, all its vertices have their neighborhood constraints satisfied, making any additional edge unnecessary. To demonstrate the theorem, we consider G^*_s, a minimum size (p,t)-sparsification of G, and construct π^* such that edges in E^*_s

Algorithm 1: *ptSpar*$(G = (V, E),\ p,\ t,\ E^{\pi})$

Input : $G = (V, E)$ a simple Graph, t an integer, $p : N \to [0, 1]$, E^{π} an ordering of E
Output : $G_s = (V_s, E_s)$ a sparsified graph

$G_s = (V_s, E_s) \leftarrow (V, \emptyset)$;
$G' = (V', E') \leftarrow (V, \emptyset)$;
for $e = uv \in E^{\pi}$ **do**
  $E' \leftarrow E' \cup \{uv\}$;
  insert $\leftarrow False$;
  $N^1_{G'}(u) \leftarrow$ direct neighbors of node u in G';
  $N^1_{G'}(v) \leftarrow$ direct neighbors of node v in G';
  for $i = 1$ *to* t **do**
    $N^i_{Gs}(u) \leftarrow$ neighbors of node u in G_s within at most i-hops;
    $N^i_{Gs}(v) \leftarrow$ neighbors of node v in graph G_s within at most i-hops;
    if $|N^i_{Gs}(u) \cap N_{G'}1(u)| < p(i)|N^1_{G'}(u)|$ *or* $|N^i_{Gs}(v) \cap N^1_{G'}(v)| < p(i)|N^1_{G'}(v)|$ **then**
      insert $\leftarrow True$;
      Break;
    end
  end
  if *insert* **then**
    $E_s \leftarrow E_s \cup \{uv\}$;
  end
end

are processed before those in $E - E^*_s$. Under this ordering, once *ptSpar* processes all edges in E^*_s, it will reject any remaining edges, resulting in G^*_s as the output, proving the theorem.

To improve the sparsification performance of the *ptSpar* algorithm, we propose three sub-optimal orders in the following subsections. Our aim is to approximate the ideal edge processing order thereby enhancing the sparsification effectiveness of the *ptSpar* algorithm. By exploring various edge ordering strategies, we seek to balance computational efficiency with the quality of the resultant sparsified graph. The first sub-optimal order relies on the ILP formulation of (p, t)-sparsification, the second one is based on edge centrality and the third one relies on a meta-heuristic to find the best order.

Linear programming based edge order: To obtain this order, we rely on our ILP formulation of (p, t)-sparsification to compute a weight for each edge that will reflect the importance of keeping it or not in the resulting sparsified graph. For this, we relax the ILP problem into an LP problem, solvable in polynomial time, by allowing variables x_e, $e \in E$, and x_w, $w \in \mathcal{W}$, to be any real values between 0 and 1. The interpretation we give to the resolution of this LP problem is that the higher the value of x_e, the more likely we want to keep e in our solution. In reverse, the lower the value of x_e, the more likely we want to remove the edge e. Thus, we can use the values of x_e, $e \in E$, as a weight and obtain an ordering for the edges. Algorithm 2 formalizes the computation of this order.

Algorithm 2: LP Ordering

Input : $G = (V, E)$ a Graph, t an integer, $p : N \to [0, 1]$
Output : Linear Programming based edge order E^{LP}

Solve the LP Relaxed problem to compute the edge scores x_e;
$E^{LP} \leftarrow$ sort edges E in descending order according to x_e;

Edge centrality based order: In this section, we propose another edge ordering that can be computed much faster than the LP order. The idea is to first process the edges with a high centrality value. Centrality is a common measure for the importance of a node or an edge in a graph. The centrality we consider here is a relaxation of local edge betweenness defined in [10]. An edge with a high edge betweenness centrality represents a bridge-like connector between two parts of a graph, the removal of which may affect the shortest paths between these parts. The local edge betweenness of an edge e is the number of shortest paths running along e, the length of which is less than or equal to some constant t. In our proposed metric, We consider all paths with a length at most t, not just the shortest paths. Furthermore, we focus only on paths directly associated with an edge in the edge set E. For every edge e, we calculate a centrality score $s(e)$ using Equation 2. In this equation, $\sigma_t(u, v|e)$ denotes the number of paths of length at most t that traverse edge e, and connect two nodes u and v that are directly linked by the edge uv. This approach ensures that the centrality score, we propose, accurately reflects the significance of each edge in connecting directly adjacent

nodes within the network.

Once all scores are computed, we sort the edges in descending order according to their score $s(e)$ to obtain an edge ordering. Algorithm 3 formalizes the computation of this ordering.

$$s(e) = \sum_{uv \in E} \sigma_t(u, v|e) \ \forall uv \in E \tag{2}$$

Algorithm 3: Edge centrality based order

Input : $G = (V, E)$ input Graph
Output : Centrality based edge order E^c
1 **for** $e \in E$ **do**
2 | compute the score s(e) using Equation 2;
3 **end**
4 $E^c \leftarrow$ sort the edges of G in descending order according to $s(e)$;

Meta-heuristic based order: In the previous two subsections, we have proposed two greedy edge orderings to improve the sparsification performance of the *ptSpar* algorithm. However, the drawback of these two solutions is that they are more time-consuming than a random edge ordering, as we will reveal in the next section with the experimental evaluation. Moreover, the computation time cannot be controlled by the user since the computation of both orderings, i.e., the LP ordering and the edge centrality based ordering, deliver the edge ordering at the end of the computation and no-intermediary results is obtained if the computation is stopped before it terminates. To overcome this problem, we explore the potential of meta-heuristics as a powerful tool for searching large solution spaces, which is essential when dealing with complex graph structures and the various constraints of the sparsification problem. Simulated Annealing (SA) [30] is one of these meta-heuristics that has the ability to escape local optima by probabilistically accepting worse solutions, thus allowing a broader exploration of the solution space. This feature makes SA an ideal candidate for finding near-optimal edge orderings. Another advantage of this solution is that the computation time can be controlled by the user by adjusting the number of SA iterations.

The Simulated Annealing(SA)-based ordering is detailed in Algorithm 4. The algorithm begins by initializing a sequence S with a random arrangement of the edges from the graph $G = (V, E)$ (cf. line 1). This sequence is the starting point for the simulated annealing process. Then, it sets the initial 'temperature' T to a predefined value T_0 (cf. line 2). This temperature is crucial in the SA technique, as it allows the acceptance of suboptimal solutions, particularly in the early stages, to ensure a comprehensive exploration of the solution space. Then, we call the *ptSpar* algorithm on G using the current edge order S, resulting in a temporary sparsified graph $G_t = (V_t, E_t)$ (cf. line 3). The number of edges in E_t is recorded in $COST_{best}$ (cf. line 4), tracking the best (i.e., smallest) edge count found in the sparsified graph so far. $COST_S$ is initialized with the size of E_t, representing the cost of the current solution (cf. line 5). In the main iterative loop of the algorithm, running for N iterations, each iteration modifies the current solution slightly by creating a new edge order S_2 from S by swapping two randomly selected edges (cf. lines 6 and 7). Then, we evaluate the new edge order S_2 (cf. lines 8 to 16) by calling the *ptSpar* algorithm again to decide whether to accept this new order based on the size of the resulting sparsified graph and the current temperature T. The temperature T is updated by multiplying it with a decreasing factor α, gradually lowering the likelihood of accepting worse solutions as the algorithm progresses (cf. line 17). Finally, the algorithm returns E^{best}, the best edge order found within the gien number of iterations (cf. line 18).

Figure 2 illustrates the results of the (p, t)-sparsification algorithms on our running example (cf. Figure 2 (a)) with parameters $t = 2$, $p(1) = 50\%$ and $p(2) = 100\%$. For each algorithm, the retained edges are in blue and the removed ones in dashed grey. The exact algorithm produces the smallest possible sparsified graph with exactly 13 edges (cf. Figure 2 (b)). The *ptSpar* algorithm used with the LP-based ordering (cf. Figure 2 (c)) and with the centrality-based ordering (cf. Figure 2 (d)) produces a near-optimal sparsification with 14 edges, which is very close to the theoretical optimal of 13 edges. It is important to note that although both orderings yield sparsifications with the same number of edges, the actual edges retained in each method are not the same resulting in different sparsified graphs. Additionally, see that the SA algorithm succeeds to produce the optimal sparsification with 13 edges (cf. Figure 2 (e) and Figure 2 (f)) if it is used with a sufficient number of iterations.

4.3 Complexity analysis

In this section, we analyze the time complexity of all the algorithms presented in the previous section. The theorems and their detailed proofs are provided in the supplementary material.

Algorithm 4: Computing the best order with simulated annealing

Input : $G = (V, E)$ a Graph, t an integer, $p : N \to [0, 1]$, N an integer (Number of iterations), T_0 a double (Initial temperature), α a double (decreasing factor)
Output : E^{best} the best edge ordering with N iterations

$S \leftarrow$ Random order of E;
$T \leftarrow T_0$;
$G_t(V_t, E_t) \leftarrow ptSpar(\ G,t,p,S)$;
$COST_{best} \leftarrow |E_t|$;
$COST_S \leftarrow |E_t|$;
for $i = 1$ *to* N **do**
  $S_2 \leftarrow$ Perturbing S by swapping the order of two random edges;
  $G_t(V_t, E_t) \leftarrow ptSpar(G,t,p,S_2)$;
  if $|E_t| < COST_{best}$ **then**
    $E^{best} \leftarrow S$;
    $COST_{best} \leftarrow |E_t|$;
  end
  if $|E_t| < COST_S$ **then**
    $S \leftarrow S_2$;
    $C_S \leftarrow |E_t|$;
  end
  else
    $r \leftarrow$ random number between 0 and 1;
    if $\exp(\frac{COST_S - |E_t|}{T}) > r$ **then**
      $S \leftarrow S_2$;
      $COST_S \leftarrow |E_t|$;
    end
  end
  $T \leftarrow \alpha * T$;
end
return E^{best};

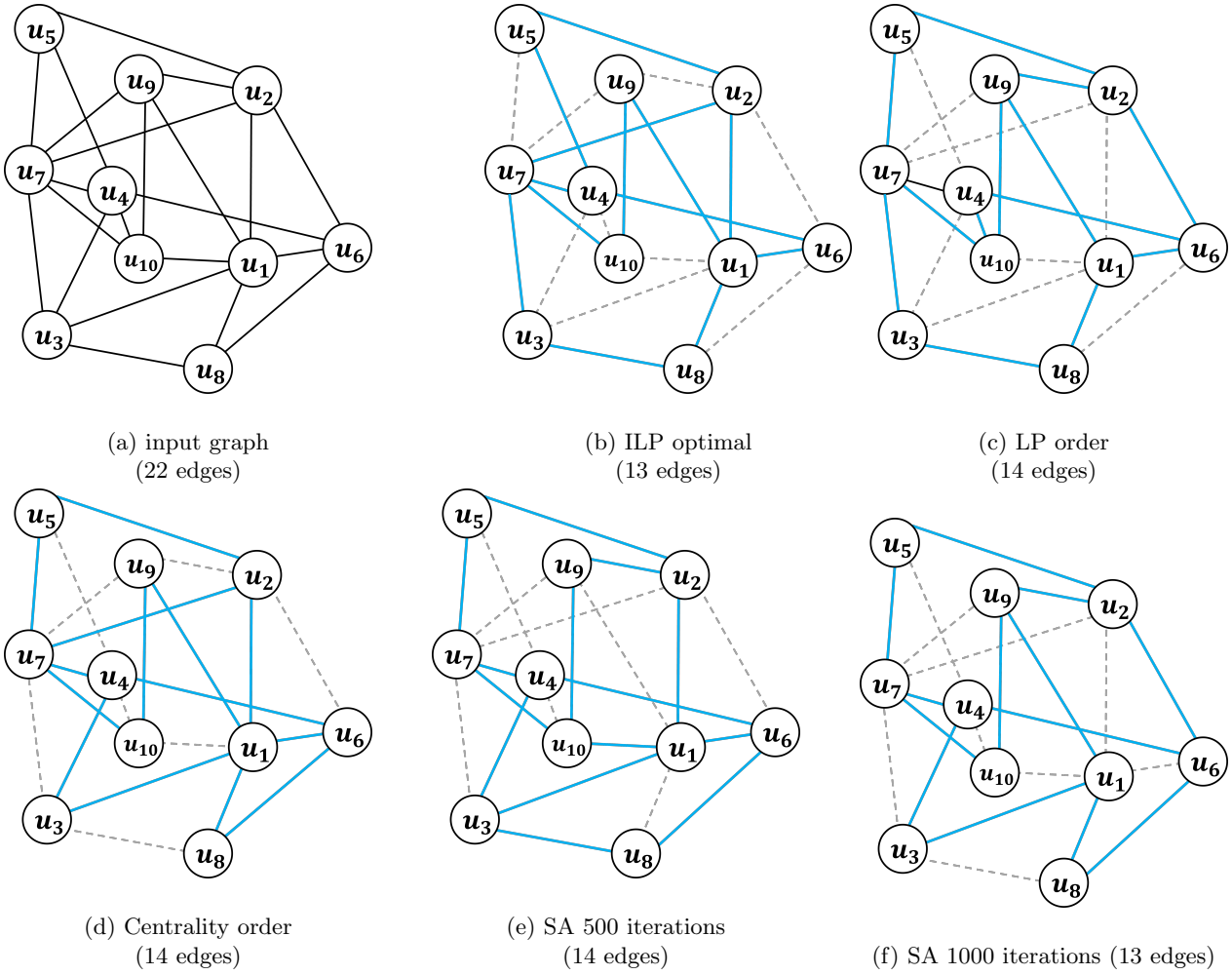
The time complexity of the *ptSpar* algorithm is $O(|E|d^t)$, where d represents the average degree in the graph G. In scenarios where every node is connected to every other node, forming a complete graph, this complexity escalates significantly, reaching $O(|E||V|^t)$.

The time complexity of the LP-based Edge Ordering algorithm is primarily derived from its linear programming (LP) problem formulation. This formulation encompasses variables representing each edge, denoted as x_e, and path variables influenced by the number of paths up to length $t-1$ starting from each vertex. Given a graph with an average degree d, the total number of these path variables scales with $|V|d^{t-1}$, where $|V|$ is the number of vertices. Consequently, the LP problem consists of $O(|E| + |V|d^{t-1})$ variables, leading to a time complexity of $O(poly(|E| + |V|d^{t-1}))$ for the algorithm, where $|E|$ signifies the number of edges.

The average time complexity of the Centrality based-Ordering is $O(|E|(d^t + log(|E|)))$. This complexity involves computing the score $s(e)$ for each edge e, which is equal to the complexity of listing all paths of lengths $\leq t$ for each pair of nodes $(u, v) \in E$. The average number of paths of length $\leq t$ between two nodes (u, v) and starting from u is of order $O(d^t)$, where d is the average degree. The number of edges is $|E|$. Therefore, the time complexity of computing all the scores $s(e)$ is $O(|E|d^t)$. We add to this, the complexity of sorting all the scores which is $O(|E|log|E|)$. In the worst case (i.e., complete graph), the time complexity would be $O(|E|(V^t + log(|E|)))$.

The average time complexity of the centrality based-Ordering is $O(|E|(d^t + \log(|E|)))$. This complexity involves computing the score $s(e)$ for each edge e and sorting them. The complexity of computing the edge scores is equal to the complexity of listing all paths of length $\leq t$ for each pair of connected nodes, i.e., for each edge in E. The average number of paths of length $\leq t$ between two connected nodes u and v and starting from u is of order $O(d^t)$, where d is the average degree. The number of edges is $|E|$. Therefore, the time complexity of computing all the scores $s(e)$ is $O(|E|d^t)$. In addition, the complexity of sorting all the scores costs $O(|E| \log |E|)$. In the worst case (complete graph), the time complexity would be $O(|E|(V^t + \log(|E|)))$.

The time complexity of the Simulated Annealing (SA)-based ordering is influenced by the number of iterations N, the initial temperature T_0, and the temperature decreasing factor α. Each iteration involves a perturbation of the edge order and a re-evaluation of the sparsification, leading to a complexity that is also



(a) input graph (22 edges)

(b) ILP optimal (13 edges)

(c) LP order (14 edges)

(d) Centrality order (14 edges)

(e) SA 500 iterations (14 edges)

(f) SA 1000 iterations (13 edges)

Figure 2: Results of the (p, t)-sparsification algorithms on the running example.

dependent on the efficiency of the *ptSpar* algorithm used within it.

5 Experimental Analysis

In this section, we present an experimental analysis of our sparsification approach. First, we evaluate the performance of the *ptSpar* algorithm provided to compute the sparsification. Then, we provide an analysis of the sensitivity of this sparsification to parameters p and t. Finally, we evaluate its effectiveness on several tasks such as shortest paths and reachability queries computation, node embedding and whole graph embedding. We also compare our sparsification with several baselines and state of the art methods to show its effectiveness. All the experiments are carried-out on an Intel core $i7$ processor with 64 Gigabytes of memory. The source code of our algorithms is available at https://gitlab.liris.cnrs.fr/coregraphie/ptspar.

Algorithms. We used the following baselines in our comparative study:

- Random Edge (RE): RE randomly eliminates a given percentage of edges.
- Local Degree (LD) [12]: LD retains the top $deg(v)^{\alpha}$ edges for each node $v \in V$, where $\alpha \in [0, 1]$
- Edge Forest Fire (EFF) [12]: EFF is based on the Forest Fire node sampling algorithm [17]. It initiates a fire at a random node and burns approximately $p/(1-p)$ neighbors, where p represents the probability threshold for burning a neighbor. Burnt neighbors are enqueued for subsequent fire initiation. EFF prunes edges based on the frequency of edge visits.
- Algebraic Distance (AD) [5]: AD uses random walk distance to compute the algebraic distance $\alpha(u, v)$ between two nodes. A low algebraic distance implies a high likelihood that a random walk starting

from u will reach v within a small number of steps. It assigns an edge score of $1 - \alpha(u, v)$ to prioritize short-range edges.

- L-Spar (LS) [27]: LS employs the Jaccard similarity function on the adjacency lists of nodes u and v to determine the edge score of (u, v). It ranks edges locally (with respect to each node) and prunes them based on their ranks.
- Simmelian Backbone (SB) [22] : SB calculates weights by counting how many triangles each edge is part of, and then retains only those edges that form the most triangles, indicating strong and interconnected relationships in the graph. During sparsification, SB removes the lower-ranked edges of each node using a specified edge-prune ratio.
- Quadrilateral Simmelian Backbone (QSB) [23]: QSB measures the Simmelianness weight of each edge (u, v) by taking into account the shared quadrangles of u and v. It follows the same pruning strategy as SB.
- Salient backbone (SLB) [9] : SLB sparsifies a graph using the disparity filter which consists in calculating a statistical significance (p-value) for each edge based on its weight and the total weight of all edges connected to the same node. Edges with p-values below a certain threshold are retained, forming the "backbone" of the graph. The rest, considered less important, are discarded.
- SparRL [32], a deep reinforcement learning-based method, sparsifies a graph by formulating the process as a Partially Observable Markov Decision Process (POMDP). It starts with a graph and at each step, chooses an edge to prune based on a policy learned from a Double DQN network. The policy is trained to maximize a reward function that encourages the preservation of certain graph properties.

Datasets. Table 2 summarizes the properties of the various datasets that we use in our extensive experiments. The synthetic graphs are used mainly to study the different ordering solutions we provided for the *ptSpar* algorithm. The real graphs are chosen according to the use cases on which we evaluated the usefulness of the obtained sparsified graphs. In fact, for each application of our sparsification, we use the most used datsets for its evaluation.

Table 2: Characteristics of datasets used in our experiments.

Name	#graphs	$\|V\|$	$\|E\|$	Use case
BLOG-CATALOG	1	$10.31K$	$333.98K$	MLNC/SP
CA-ASTROPH	1	$18.77K$	$198.11K$	SP
CA-HEPTH	1	$9.8K$	$25.9K$	SP
CiteSeer	1	$3.2K$	$4.5K$	NC/SP
COLLAB	5000	$372.5K$	$49.1M$	GC
Cora	1	$2.7K$	$5.4K$	NC/SP
ENZYMES	600	$19.5K$	$74.6K$	GC
FLICKR	1	$89K$	$899K$	NC/SP
FLICKR-Large	1	$80.51K$	$5.89M$	MLNC
IMDB-BINARY	1000	$19.77K$	$96.53K$	GC
MSRC-21C	209	$8.4K$	$20.2K$	GC
PROTEINS	1113	$43.5K$	$162.1K$	GC
PubMed	1	$19.7K$	$44.3K$	NC/SP
SYNTH1	30	20	60	RT
SYNTH2	30	50	350	RT
SYNTH3	30	100	$1.4K$	RT

MLNC: Multi-label Node Classification, SP: Shortest Paths,
NC: Node Classification, GC: Graph Classification, RT: Running Time.

Metrics. We use the following metrics:

- Sparsification runtime measured in seconds,
- Sparsification ratio that represents the ratio of the number of deleted edges over the total number of edges (see Equation 1), and
- Entropy loss to measure the information loss after the sparsification. The graph entropy is a measure of the structural information of a graph and serves as a complexity measure [7]. Given a graph $G(V, E)$,

the Shanon entropy of G (i.e., $I(G)$) is computed as follows [7]:

$$I(G) = -\sum_{u \in V} \frac{deg(u)}{\sum_{u \in V} deg(u)} \log\left(\frac{deg(u)}{\sum_{u \in V} deg(u)}\right) \tag{3}$$

The entropy loss is the normalized difference between the entropy of the original graph and the entropy of the sparsified graph. Let G be the original graph and G_s be the sparsified graph, we compute the entropy loss as follows:

$$E_{loss} = \frac{|I(G) - I(G_s)|}{I(G)} \tag{4}$$

Note that the lower is the entropy loss the better is the sparsification.

5.1 Evaluating the edge ordering methods

In this subsection, we present a comparative experimental study of the edge orderings we considered for optimizing the *ptSpar* algorithm. These orderings are: LP ordering, centrality based ordering and the Simulated-annealing (SA) ordering. We compare them with a random ordering of the edges. The aim of these experiments is to show that the performance in term of sparsification ratio can be improved by considering different edge orderings. For this experiment, we use 3 families of synthetic graphs and the following sparsification parameters $t = 2$, $p(1) = 0.0$ and $p(2) = 0.5$. For a reliable and accurate comparison, we carried-out around thirty tests on each family of graphs for each edge ordering solution. The results of the comparison are depicted in Table 3. Note that the user configuration of the SA is $T_0 = 10$, $N = 1000$ and $\alpha = 0.99$. We notice that the two greedy orderings, LP and centrality, and the SA algorithm outperform the random ordering of edges in terms of sparsification performance. The results clearly show that the centrality and the SA orderings are the best algorithms. The centrality ordering seems really interesting and offers the best trade-off between sparsification performance and runtime. However, we can see that the *ptSpar* algorithm with a random order of edges is much faster than with the other orderings methods. Therefore, we will be using it in the rest of the experiments.

Table 3: Evaluation of the *ptSpar* algorithm with different edge orderings.

		SYNTH1	SYNTH2	SYNTH3
Random	avg $\|E_s\|$	28	121.6	367
	avg time	0.001	0.008	0.05
LP	avg $\|E_s\|$	25.24	113.26	354.3
	avg time	0.02	2.5	212
Centrality	avg $\|E_s\|$	23.55	105.66	323.2
	avg time	0.01	0.02	0.09
SA	avg $\|E_s\|$	21.56	105.9	340.4
	avg time	0.5	5.2	40

5.2 Evaluating the impact of the sparsification parameters p and t

In this series of experiments, we study the effect of parameters p and t on the sparsification performance.

As mentioned before, our sparsification allows users to control the trade-off between information loss and sparsification ratio (i.e., space gain). To do so, the user varies the parameters p and t according to its needs (available memory and the targeted use case) to find the configuration that suits him. The ideal scenario is to minimize the information loss (entropy loss) while maximizing the sparsification ratio. Table 4 gives the sparsification ratio and entropy loss obtained by our sparsification on the CA-AstroPh dataset, while varying the neighborhood preservation proportion p. We set $p(t) = 1$ in all experiments, which means that the whole initial neighborhood of each node can be retrieved in a neighborhood of radius $r = t$ at maximum. This ensures that reachability queries are fully preserved for all vertices. As expected, the sparsification ratio decreases (and the entropy loss increases) as the preserved proportion of neighborhood increases and vice-versa. Some of the values of the sparsification ratio obtained with the various combinations of parameters are very satisfactory. The same holds for the entropy loss (max value $< 5\%$). In addition, we remark that the sparsification ratio range is wide (from 7% to 75%) which confirms the possibility of controlling effectively the trade-off information loss/sparsification ratio using parameters p and t. The choice of the best configuration of parameters depends essentially on the nature of the graph to be sparsified and the user needs. Particularly, for this example, the configurations ($t = 2,\ p = (0.5, 1)$) and ($t = 3,\ p = (0.5, 0.7, 1)$) seem interesting and

are a good trade-off between sparsification ratio and entropy loss with a sparsification ratio $> 45\%$ and an entropy loss $< 1\%$.

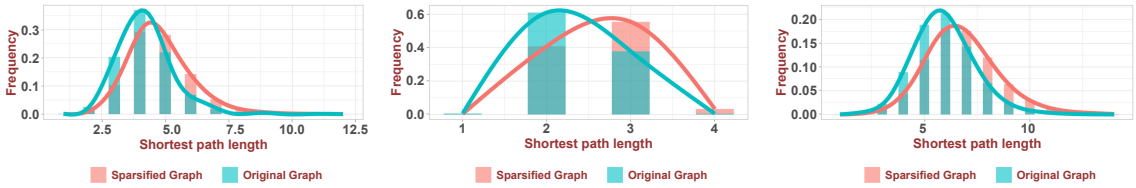
Table 4: Sparsification ratio vs entropy loss of the Ca-AstroPh dataset with different combinations of parameters p and t.

t	$p(1)$	$p(2)$	$p(3)$	sparsification ratio	Entropy loss
2	0.2	1.0	-	58.13%	1.71%
	0.5	1.0	-	45.82%	0.90%
	0.7	1.0	-	26.39%	0.66%
	0.9	1.0	-	7.43%	0.31%
3	0.0	0.2	1.0	75.00%	4.61%
	0.2	0.5	1.0	71.50%	2.57%
	0.5	0.7	1.0	46.73%	0.85%
	0.7	0.9	1.0	26.43%	0.66%

5.3 Evaluation the distribution of the shortest path lengths with (p, t)-Sparsification

In this experiments, we show that (p, t)-sparsification allows to approximate distances (shortest path lengths) between nodes. To do so, we sparsify three unweighted undirected graphs with the following combination of parameters $t = 2$, $p(1) = 0.5$, and $p(2) = 1.0$. Then, we compute all shortest paths between all nodes.

Figure 3 shows the distribution of the shortest path lengths in the original and sparsified graphs for the three datasets. We note that the two curves have almost the same pace. This shows that our sparsification preserves the distribution of the lengths of the shortest paths on the 3 datasets. However, the curves of the sparsified graphs are slightly stretched and shifted from the original curves. This is due to the stretching of the paths as a result of sparsification. This stretch is not really considerable because of the preservation of 50% of the direct neighbors of each vertex in the graph.
It is important to note here that we cannot draw a similar distribution for the shortest paths of other baseline methods, as they do not preserve the connectivity of the graph. However, we have evaluated the increase in the length of the shortest paths and the loss of reachability queries for these methods in Section 5.5.



(a) Ca-AstroPh dataset (d) Blog-Catalog dataset (c) Ca-HepTh dataset

Figure 3: Distribution of shortest path lengths of the original and sparsified graphs.

5.4 Evaluating the information loss

In this series of experiment, we focus on evaluating the quality of the obtained sparsified graphs by measuring the loss of entropy for each method. Table 5 gives the information loss rates measured by the loss in entropy of all the sparsification methods on four datasets. For a rigorous comparison, we have selected datasets that have different densities and contain hundreds of graphs. It's worth noting that SparRL, which is a deep reinforcement learning-based method, was not applied in this context due to two main reasons: (1) It is computationally intensive and requires individual training for each graph. This is inefficient for datasets containing hundreds of graphs. (2) It has a reward function with a limited scope that does not consider entropy – a key factor in this case. The obtained results are as follows:

- On the COLLAB Dataset, the (p, t)-sparsification method proves to be the most effective, achieving the least entropy loss of 0.99%. This means that it best preserves the structure of the original graph. In contrast, the Local Degree (LD) method experiences the greatest entropy loss with 21.3%, implying substantial information loss during the sparsification.

- On the IMDB-BINARY Dataset, the Simmelian Backbone (SB) method records the smallest entropy loss with 1.1%, indicating the most successful preservation of the original graph's structure. (p,t)-sparsification is almost as good with an entropy loss of 1.45%. The LD method, once again, shows the worst entropy loss with 19.36%.
- On the MSRC_21C Dataset, the (p,t)-sparsification method excels over all the other methods by achieving the lowest entropy loss (0.6%) which denotes the best preservation of the original graph's structure. The method with the worst entropy loss is again LD, with a value of 6.4%.
- On the PROTEINS Dataset, the L-Spar (LS) method and the (p,t)-sparsification demonstrate the best performance, each achieving the best entropy loss (1.3% and 1.5% respectively).

To summarize, the (p,t)-sparsification method demonstrates consistent and effective performance across multiple datasets, showing the least entropy loss in most cases. This indicates that the (p,t)-sparsification method is capable of preserving most of the graph's original information, proving it to be the best choice for various use cases.

Table 5: Effect of the sparsification on the graph entropy

	COLLAB	IMDB-BINARY	MSRC_21C	PROTEINS
ptSpar	**1.00**%	1.50%	**0.60**%	1.50%
SLB	20.80%	22.70%	4.40%	4.10%
AD	11.80%	3.70%	1.50%	4.90%
LS	7.70%	6.40%	0.60%	**1.30**%
QSB	2.90%	1.20%	1.70%	5.30%
SB	1.70%	**1.10**%	1.80%	4.40%
EFF	7.40%	6.60%	2.90%	3.80%
LD	21.30%	19.40%	6.40%	4.80%
RE	6.80%	5.90%	2.70%	2.50%

5.5 Evaluating the usefulness of the sparsified graphs

We have applied many graph algorithms on the sparsified graphs. Our first motivation is to be able to use these algorithms directly on the sparsified graphs to reduce memory space and speed up running times. So, the purpose of the following experiments is to show the effectiveness of our sparsification in terms of speeding-up such graph algorithms, while handling large graphs and providing good approximations of the original results. For this, and for all the following experiments, we compute two new metrics in addition to the sparsification ratio namely:

- **Speed-up factor:** the ratio between the algorithm run-time on the original graph and its run-time on the sparsified graph. The higher the speed-up factor, the faster the graph algorithm on the sparsified graph.
- **Performance Preservation:** This metric measures the degree to which the performance on the sparsified graph approaches the one obtained on the original graph. In contexts like classification, it gauges the relative conservation of accuracy from the original graph to its sparsified counterpart. A higher Performance Preservation value signifies that the sparsified graph has effectively retained, or closely approximated, the properties of the original graph

Shortest Paths and Reachability Queries: To evaluate how shortest paths are impacted with sparsification, we computed the average increase in shortest path length between 10,000 pairs of nodes, chosen randomly, for various graph sparsification methods. A smaller increase is better as it indicates the sparsification process preserves the shortest path lengths in the original graph more accurately. Table 6 illustrates the obtained results. Values in parentheses represent the failure rate which is the percentage of node pairs that became disconnected (unreachable from each other) in the sparsified graph. It is worthy to note that the SparRL method couldn't be executed on the Flickr dataset due to time constraints, indicating potential scalability issues with this approach.

In terms of preserving the shortest path between randomly chosen pairs of nodes, the proposed (p,t)-sparsification technique consistently performs well across all datasets. It achieves the lowest average increase in path length for all datasets, indicating that it maintains the structural integrity of the original graphs to a great extent. Furthermore, it achieves a zero percent failure rate for three out of four datasets, which

suggests that the sparsified graphs generated using (p,t)-sparsification are highly connected, preserving the reachability of nodes effectively. The Salient Backbone method also performs competitively, producing a low increase in path length and relatively low failure rates. This suggests that it also does a good job of maintaining the structure and connectivity of the original graphs during the sparsification process. On the other hand, methods such as Algebraic Distance, Local Similarity, Quadrilateral Simmelian, and Simmelian Sparsifier exhibit a higher increase in path length and a higher failure rate across most datasets, implying a greater degree of distortion in the sparsified graphs.

Table 6: Performance on shortest paths and reachability queries.

	Cora	Citeseer	pubmed	Flickr
AD	14.1% (55.6%)	2.4% (92.8%)	19.6% (61.0%)	4.7% (36.0%)
LS	46.3% (19.6%)	44.5% (51.2%)	23.5% (5.0%)	6.3% (0.0%)
QSB	15.1% (41.4%)	15.1% (46.9%)	7.8% (39.6%)	9.2% (0.3%)
SB	15.9% (41.1%)	12.4% (46.0%)	7.3% (39.9%)	9.0% (0.3%)
EFF	8.0% (22.8%)	1.6% (22.0%)	5.5% (41.7%)	19.3% (6.0%)
LD	7.0% (2.7%)	13.9% (20.0%)	4.7% (0.4%)	4.4% (0.0%)
RD	14.1% (18.1%)	12.0% (28.9%)	8.3% (25.6%)	10.7% (0.0%)
Our	**2.7% (0.0%)**	**1.7% (0.1%)**	**2.2% (0.0%)**	**2.8% (0.0%)**
SparRL	3.3% (4.8%)	**1.1% (12.8%)**	1.8% (4.5%)	out of time
SLB	2.3%(4.1%)	1.5%(5.9%)	**1.3%(5.2%)**	**1.9%(0.5%)**

Graph kernels: Graph kernels predominantly rely on local neighborhood information of nodes. Such methods derive graph representations by delving deep into node neighborhoods and extracting pertinent features, encompassing walks, shortest paths, and other local substructures. Given that our (p,t)-sparsification meticulously retains the local neighborhood up to a radius t, a pertinent question arises: can graph kernels algorithms accelerate on sparsified graphs without significant compromise on performance? To answer this inquiry, we run a graph classification task on graph classification datasets, namely COLLAB, IMDB-BINARY, MSRC-21C, and PROTEINS. Here, we set $t = 3$, $p(1) = 0.0$, $p(2) = 0.5$ and $p(3) = 1.0$. On these sparsified datasets, we executed various graph embedding algorithms, including the Shortest Path graph kernel (SP) [3], Weisfeiler-Lehman Optimal Assignment WL-OA graph kernel [28, 15], The Neighborhood Hash NH graph kernel [13] and deep Renyi entropy graph kernel (REK) [33]. We gauged the efficacy of these algorithms on both input graph and sparsified graphs. We used SVM algorithm as classifier, with a 10-fold cross-validation, served as our performance metric. For the sake of fairness, all baseline methods maintained an identical sparsification ratio. The SparRL method was omitted from this evaluation due to its innate latency and the requisite training for each graph, proving inefficient given the multiplicity of graphs in our datasets.
Table 7 shows the performance of graph kernels on the sparsified graphs. We notice that all kernels run faster on sparsified graphs in most cases. This Kernel computation speed-up is more noticeable on denser datasets such as COLLAB. Since the sparsified graphs produced by all methods are of the same size (same sparsification ratio for fair comparison), the speed up factors are the same for all methods. However, the performance preservation of graph kernel methods on sparsified graphs provides pivotal insights into the robustness and efficacy of different sparsification approaches, particularly emphasizing our (p,t)-sparsification method. Across a diverse range of datasets, our method's performance, in many instances, either leads the cohort or remains competitively in line with the best-performing methods. For example, in the COLLAB dataset with the Shortest Path (SP) graph kernel, our method reaches a performance preservation of 100%, a performance matched only by SLB and LD, while outperforming other benchmarks such as SB, EFF, and LS. This level of consistency in preserving the integrity of the original graph structure continues across various kernels like WL, NH, and REK. What is particularly notable is the general out-performance of our method when contrasted against SB in datasets like MSRC_21C using the REK kernel, where our method achieves a perfect score of 100% versus SB's 20%. Yet, our method proves its mettle even in situations where it doesn't lead but exhibits comparable performance, such as in the case of the WL kernel in the same dataset. While LD achieved the top score of 92%, our method's 88% was closely aligned, demonstrating its competitive. However, it's worth noting that while our method isn't always the definitive leader across all datasets and kernels, it consistently ranks among the top contenders, rarely deviating far from the highest scores. The associated speed-up rates also underscore the computational advantages of our sparsification approach. To sum up, the (p,t)-sparsification method we propose serves as a powerful tool, often leading in performance preservation and, when not, still staying well ranked within the top performing methods across a wide range of datasets and graph kernels.
Node embedding: In this series of experiment, we use sparsified graphs to compute node embedding. Then, to see if the obtained embedding are as relevant as the ones computed on the full graph, we evaluate their

Table 7: Graph kernel performance on the sparsified graphs.

Dataset	Kernel	Sr	Speed up	Performance Preservation					
				EFF	LD	LS	ptSpar	SB	SLB
COLLAB	SP	91.4%	2.75	97%	**100%**	90%	**100%**	95%	**100%**
	WL		1.23	86%	**92%**	87%	88%	88%	85%
	NH		1.54	83%	**88%**	84%	87%	84%	83%
	REK		1.86	83%	78%	86%	**88%**	87%	65%
IMDB-d	SP	72.2%	1.14	96%	**100%**	81%	100%	93%	99%
	WL		1	93%	**96%**	90%	95%	89%	89%
	NH		1.11	91%	92%	83%	**94%**	89%	87%
	REK		1.46	91%	81%	93%	95%	**97%**	68%
MSRC_21C	SP	46.8%	1.04	97%	97%	89%	**100%**	89%	99%
	WL		1.24	98%	**100%**	93%	**100%**	95%	24%
	NH		1.34	97%	95%	91%	**100%**	94%	24%
	REK		1.15	95%	96%	99%	**100%**	99%	20%
PROTEINS	SP	36.1%	1.39	98%	**100%**	92%	**100%**	96%	**100%**
	WL		1.2	96%	96%	89%	**97%**	95%	94%
	NH		1.12	98%	**99%**	94%	**99%**	98%	95%
	REK		1.86	97%	**96%**	99%	**99%**	97%	80%

efficacy in two tasks: node classification and multi-label classification. We leveraged two predominant algorithms for this endeavor: the Graph Attention Network (GAT) [31] for node classification, and Node2vec [11] for multi-label classification. The experiments are conducted on graphs with distinct sparsification ratios: 45% for the multi-label task and 20% for the graph classification task. It's worth noting that, for multi-label classification on the Flickr-large dataset, we have not included the results of the Salient Backbone (SLB) method because it failed to sparsify this large graph within the time limit of 24 hours. Additionally, the SparRL method is not present in evaluations for both tasks because it is not possible to express classification within its objective function.
Table 8 presents the outcomes of the different sparsification methods. Across all datasets, we can see that (p, t)sparsification consistently excelled. On datasets such as 'PROTEINS', (p, t)sparsification achieves an almost impeccable accuracy preservation rate of 99.68%. Similarly, in the 'Cora' and 'Flickr' datasets, it achieves the impressive rates of 96.97% and 99.16%. These figures attest to the technique's proficiency in preserving critical graph structures essential for GAT. Local Degree Sparsifier and Local Similarity also delivered interesting outcomes in certain datasets but Quadrilateral Simmelian and Simmelian Sparsifier reported suboptimal results, further underscoring the significance of (p, t)-sparsification's results.

Table 9 presents the results of multi-label node classification using node2vec embedding on the sparsified graphs. We can clearly see the out-performance of (p, t)-Sparsification. In the Blog Catalog dataset, (p, t)-Sparsification achieves 93.03% for Micro F1 and 90.75% for Macro F1 metrics. Its excellent performance is further underscored in the Flickr dataset, where it registers a perfect 100% in both metrics preservation. This clearly shows the robustness of (p, t)-Sparsification in retaining crucial graph properties vital for node2vec embedding. The other methods have much less effective results.

Table 8: Performance of Node classification on sparsified graphs.

Method	Cora	Citeseer	pubmed	Flickr
AD	90.47%	87.07%	89.32%	92.15%
EFF	71.17%	55.51%	81.87%	99.01%
LD	96.80%	**96.75%**	99.65%	99.58%
LS	94.81%	96.26%	97.94%	97.42%
ptSpar	**96.97%**	94.53%	**99.68%**	**99.16%**
QSB	46.66%	36.90%	52.59%	93.08%
RE	90.47%	85.10%	88.31%	94.02%
SB	47.27%	37.34%	53.76%	93.56%
SLB	87.3%	78.40%	88.83%	92.72%

Table 9: Performance of multi-label node classification on sparsified graphs.

Method	Blog Catalog		Flickr	
	Micro F1 %	Macro F1 %	Micro F1 %	Macro F1 %
AD	35.84 %	15.23 %	43.11 %	28.6 %
EFF	36.48 %	16.61 %	44.01 %	29.6%
LD	37.74 %	14.56 %	44.22 %	28.2%
LS	35.46 %	16.69 %	44.43 %	29.6 %
ptSpar	**93.03 %**	**90.75** %	**100%**	**100%**
QSB	36.54 %	14.31 %	45.86 %	31.5 %
RE	35.37 %	13.05%	44.70 %	45%
SB	38.35 %	14.35 %	44.49%	24.9%

6 Conclusion and future work

In this paper, we presented a graph sparsification approach designed to produce a graph skeleton that can be used instead of the original large graph as input in many graph analysis algorithms. To do so, our sparsification controls the amount of neighborhood information preserved in the resulting sparsified graph with two parameters: a function p that gives the proportion of each node's original neighbors to be preserved in its i-hops neighborhood in the sparsified graph, and a threshold t for which p reaches its maximal value. We also presented several algorithms to compute this sparsification with the minimum cost, and showed their effectiveness in sparsifying input graphs through an extensive experimental evaluation on multiple real-life as well as synthetic graph datasets. Furthermore, We showed that the skeletons computed by the proposed approach can be used without any addition or de-sparsification as input to multiple graph applications, such as node embedding, graph classification, and shortest path approximations, with interesting trade-offs between algorithm runtime speed-up and precision loss.

As for future work, we consider a more thorough analysis of (p, t)-sparsification impact on walk based graph learning algorithms such as Node2vec and DeepWalk. In fact, we observed some situations where the learning accuracy increased when the graph was sparsified. This was a quite unexpected observation. While we guess that walks are biased in the right direction by removing edges, characterizing such edges remains an open question. Another important open question is to find an efficient method to order graph edges. This would allow us to significantly improve the time complexity of the approach. In addition, we aim to design an incremental version of our sparsification to deal with dynamic graphs or graph streams.
We note also that our approach can be used on both directed and undirected graphs.

Acknowledgment

This work was supported by Agence Nationale de la Rechetche (ANR) under grant number ANR-20-CE23-0002. The authors thank Walid MEGHERBI for his valuable help in the experiments conducted for the final version of the paper.

References

[1] András A. Benczúr and David R. Karger. Approximating s-t minimum cuts in $o(n^2)$ time. In *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing*, STOC '96, page 47–55, New York, NY, USA, 1996. Association for Computing Machinery.

[2] Francesco Bonchi, Gianmarco De Francisci Morales, Aristides Gionis, and Antti Ukkonen. Activity preserving graph simplification. *Data Mining and Knowledge Discovery*, 27(3):321–343, November 2013.

[3] Karsten M Borgwardt and Hans-Peter Kriegel. Shortest-path kernels on graphs. In *Fifth IEEE international conference on data mining (ICDM'05)*, pages 8–pp. IEEE, 2005.

[4] Silvia Butti and Stanislav Živný. Sparsification of binary csps. *SIAM Journal on Discrete Mathematics*, 34(1):825–842, 2020.

[5] Jie Chen and Ilya Safro. Algebraic distance on graphs. *SIAM Journal on Scientific Computing*, 33(6):3468–3490, 2011.

[6] Michele Coscia and Frank MH Neffke. Network backboning with noisy data. In *2017 IEEE 33rd International Conference on Data Engineering (ICDE)*, pages 425–436. IEEE, 2017.

[7] Matthias Dehmer and Abbe Mowshowitz. A history of graph entropy measures. *Information Sciences*, 181(1):57–78, 2011.

[8] David Durfee, John Peebles, Richard Peng, and Anup B. Rao. Determinant-preserving sparsification of sddm matrices. *SIAM Journal on Computing*, 49(4):FOCS17–350–FOCS17–408, 2020.

[9] Daniel Grady, Christian Thiemann, and Dirk Brockmann. Robust classification of salient links in complex networks. *Nature communications*, 3(1):1–10, 2012.

[10] Steve Gregory. A fast algorithm to find overlapping communities in networks. In Walter Daelemans, Bart Goethals, and Katharina Morik, editors, *Machine Learning and Knowledge Discovery in Databases*, pages 408–423, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[11] Aditya Grover and Jure Leskovec. node2vec: Scalable feature learning for networks. In *Proceedings of the 22nd ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 855–864, 2016.

[12] Michael Hamann, Gerd Lindner, Henning Meyerhenke, Christian L Staudt, and Dorothea Wagner. Structure-preserving sparsification methods for social networks. *Social Network Analysis and Mining*, 6:1–22, 2016.

[13] Shohei Hido and Hisashi Kashima. A linear-time graph kernel. In *2009 Ninth IEEE International Conference on Data Mining*, pages 179–188. IEEE, 2009.

[14] Dong-Hee Kim, Jae Dong Noh, and Hawoong Jeong. Scale-free trees: The skeletons of complex networks. *Physical Review E*, 70(4):046126, 2004.

[15] Nils M Kriege, Pierre-Louis Giscard, and Richard Wilson. On valid optimal assignment kernels and applications to graph classification. *Advances in neural information processing systems*, 29, 2016.

[16] Sofiane Lagraa, Hamida Seba, Riadh Khennoufa, Abir MBaya, and Hamamache Kheddouci. A distance measure for large graphs based on prime graphs. *Pattern Recognition*, 47(9):2993–3005, 2014.

[17] Jure Leskovec and Christos Faloutsos. Sampling from large graphs. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 631–636, 2006.

[18] Yuzhi Liang, Chen chen, Yukun Wang, Kai Lei, Min Yang, and Ziyu Lyu. Reachability preserving compression for dynamic graph. *Information Sciences*, 520:232–249, 2020.

[19] Y. Liu, T. Safavi, A. Dighe, and D. Koutra. Graph summarization methods and applications: A survey. *ACM Comput. Surv.*, 51(3):62:1–62:34, June 2018.

[20] Yike Liu, Tara Safavi, Abhilash Dighe, and Danai Koutra. Graph summarization methods and applications: A survey. *ACM computing surveys (CSUR)*, 51(3):1–34, 2018.

[21] Michael Mathioudakis, Francesco Bonchi, Carlos Castillo, Aristides Gionis, and Antti Ukkonen. Sparsification of influence networks. In *Proceedings of the 17th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '11, page 529–537, New York, NY, USA, 2011. Association for Computing Machinery.

[22] Bobo Nick, Conrad Lee, Pádraig Cunningham, and Ulrik Brandes. Simmelian backbones: Amplifying hidden homophily in facebook networks. In *Proceedings of the 2013 IEEE/ACM international conference on advances in social networks analysis and mining*, pages 525–532, 2013.

[23] Arlind Nocaj, Mark Ortmann, and Ulrik Brandes. Untangling hairballs: From 3 to 14 degrees of separation. In *International symposium on graph drawing*, pages 101–112. Springer, 2014.

[24] L. Paul Chew. There are planar graphs almost as good as the complete graph. *Journal of Computer and System Sciences*, 39(2):205 – 219, 1989.

[25] David Peleg and Alejandro A. Schäffer. Graph spanners. *Journal of Graph Theory*, 13(1):99–116, 1989.

[26] Sherif Sakr, Angela Bonifati, Hannes Voigt, Alexandru Iosup, Khaled Ammar, Renzo Angles, Walid G. Aref, Marcelo Arenas, Maciej Besta, Peter A. Boncz, Khuzaima Daudjee, Emanuele Della Valle, Stefania Dumbrava, Olaf Hartig, Bernhard Haslhofer, Tim Hegeman, Jan Hidders, Katja Hose, Adriana Iamnitchi, Vasiliki Kalavri, Hugo Kapp, Wim Martens, M. Tamer Özsu, Eric Peukert, Stefan Plantikow, Mohamed Ragab, Matei Ripeanu, Semih Salihoglu, Christian Schulz, Petra Selmer, Juan F. Sequeda, Joshua Shinavier, Gábor Szárnyas, Riccardo Tommasini, Antonino Tumeo, Alexandru Uta, Ana Lucia Varbanescu, Hsiang-Yun Wu, Nikolay Yakovets, Da Yan, and Eiko Yoneki. The future is big graphs: a community view on graph processing systems. *Commun. ACM*, 64(9):62–71, 2021.

[27] Venu Satuluri, Srinivasan Parthasarathy, and Yiye Ruan. Local graph sparsification for scalable clustering. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 721–732, 2011.

[28] Nino Shervashidze, Pascal Schweitzer, Erik Jan Van Leeuwen, Kurt Mehlhorn, and Karsten M Borgwardt. Weisfeiler-lehman graph kernels. *Journal of Machine Learning Research*, 12(9), 2011.

[29] Daniel A. Spielman and Shang-Hua Teng. Spectral sparsification of graphs. *SIAM Journal on Computing*, 40(4):981–1025, 2011.

[30] Peter JM Van Laarhoven and Emile HL Aarts. Simulated annealing. In *Simulated annealing: Theory and applications*, pages 7–15. Springer, 1987.

[31] Petar Velickovic, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, Yoshua Bengio, et al. Graph attention networks. *stat*, 1050(20):10–48550, 2017.

[32] R. Wickman, X. Zhang, and W. Li. A generic graph sparsification framework using deep reinforcement learning. In *2022 IEEE International Conference on Data Mining (ICDM)*, pages 1221–1226, Los Alamitos, CA, USA, dec 2022. IEEE Computer Society.

[33] Lixiang Xu, Lu Bai, Xiaoyi Jiang, Ming Tan, Daoqiang Zhang, and Bin Luo. Deep rényi entropy graph kernel. *Pattern Recognition*, 111:107668, 2021.

[34] Ronda J Zhang, H Eugene Stanley, and Fred Y Ye. Extracting h-backbone as a core structure in weighted networks. *Scientific reports*, 8(1):1–7, 2018.