universität wien

DISSERTATION | DOCTORAL THESIS

Titel | Title

Efficient and Expressive Graph Learning

verfasst von | submitted by

Franka Regina Bause B.Sc. M.Sc.

angestrebter akademischer Grad | in partial fulfilment of the requirements for the degree of

Doktorin der technischen Wissenschaften (Dr.techn.)

Wien | Vienna, 2025

Studienkennzahl lt. Studienblatt | Degree programme code as it appears on the student record sheet:	UA 786 880
Dissertationsgebiet lt. Studienblatt | Field of study as it appears on the student record sheet:	Informatik
Betreut von | Supervisor:	Assoz. Prof. Dipl.-Inf. Dr. Nils Morten Kriege

Acknowledgments

Completing this dissertation would not have been possible without the support and encouragement of many people, whom I would like to thank here.

First, I would like to express my deepest gratitude to my supervisor, Prof. Nils Kriege, for giving me the opportunity to work in his group and for his guidance throughout my research journey. Attending one of his seminars during my undergraduate studies sparked my initial interest in graph theory.

He always had time to discuss ideas, provided valuable and constructive feedback, and fostered a welcoming group environment. I am especially grateful for the freedom he gave me to pursue my ideas while also including me in many research projects, helping me build my network. I feel incredibly fortunate to have him as my supervisor.

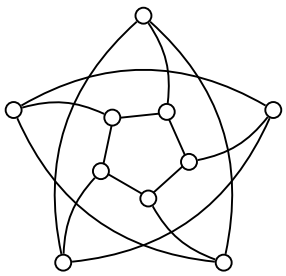
I also want to thank the reviewers of this thesis, Yllka Velaj and Bastian Rieck, for agreeing to read and evaluate my work. Your time and thoughtful feedback are greatly appreciated.

Thank you to the entire Data Mining and Machine Learning group for the supportive community. Special thanks go to Anatol and Timo for the great time in the office, Anna for being an awesome travel companion, Lukas and Lena for their advice, and Pascal and Martin for all the running. I am grateful to my co-authors for their collaboration and expertise, Ewald for keeping the servers running smoothly, and Charlotte for always handling organizational matters and administrative tasks.

Thanks also to the Research Unit Machine Learning at TU Wien, particularly to Pascal for his revising skills and TikZ magic, Fabian for the fun discussions, Max for his many ideas, Tamara for the event collaborations, and Thomas Gärtner for supporting our ideas.

To my friends and family, thank you for your encouragement and support. Finally, I thank Martin for believing in me and listening to many presentation rehearsals.

To all of you — thank you so much!



Bibliographic Note

The results of this thesis were published in conference proceedings and journal articles. Therefore, the chapters of this thesis are based on the following publications, which are also included in Appendix A:

- **Paper A:** Franka Bause, David B. Blumenthal, Erich Schubert, and Nils M. Kriege. *Metric Indexing for Graph Similarity Search.* In: Similarity Search and Applications (SISAP), 2021.
- **Paper B**[1]**:** Franka Bause, Erich Schubert, and Nils M. Kriege. *EmbAssi: Embedding Assignment Costs for Similarity Search in Large Graph Databases.* In: Data Mining and Knowledge Discovery, 2022.
- **Paper C:** Franka Bause*, Christian Permann*, and Nils M. Kriege. *Approximating the Graph Edit Distance with Compact Neighborhood Representations.* In: Machine Learning and Knowledge Discovery in Databases. Research Track (ECML PKDD), 2024.
- **Paper D:** Franka Bause and Nils M. Kriege. *Gradual Weisfeiler-Leman: Slow and Steady Wins the Race.* In: The First Learning on Graphs Conference (LoG), 2022.
- **Paper E:** Franka Bause, Samir Moustafa, Johannes Langguth, Wilfried N. Gansterer, and Nils M. Kriege. *On the Two Sides of Redundancy in Graph Neural Networks.* In: Machine Learning and Knowledge Discovery in Databases. Research Track (ECML PKDD), 2024.
- **Paper F:** Franka Bause*, Fabian Jogl*, Patrick Indri, Tamara Drucks, David Penz, Nils M. Kriege, Thomas Gärtner, Pascal Welke, Maximilian Thiessen. *Maximally Expressive GNNs for Outerplanar Graphs.* In: Transactions on Machine Learning Research (TMLR), 2025.

*Equal contribution.

[1] Part of this contribution is based on the Master's thesis "Efficient Approximate k-Nearest-Neighbor-Search in Large Graph Databases" [Bau20]. The full information on which contributions were added during the PhD studies can be found in Appendix A.2.

Abstract

Graphs are versatile representations for real-world objects and abstract entities in many applications, such as social network analysis, healthcare, and fraud detection. Graph learning tasks, ranging from node classification and link prediction to graph classification and regression, have become more and more prevalent. With increasing data, scalability has become crucial for graph learning methods. Additionally, expressivity, which measures the ability of a method to distinguish pairs of non-isomorphic graphs, still poses a fundamental challenge in graph learning.

This thesis aims to advance graph learning techniques by enhancing scalability, improving similarity measures, and addressing expressivity limitations in current methods. Since graph similarity measures typically rely on similarity measures for nodes, we investigate and develop methods to capture the neighborhood of nodes more accurately. Specifically, we focus on efficiently approximating the graph edit distance for search in graph databases. Using an existing lower bound for the graph edit distance in combination with a metric index in a filter-verification framework, we make searching in large graph databases possible. We propose novel lower bounds that can be embedded into ℓ_1 space to be used as an even more efficient first filter. Using the distance between pruned unfolding trees as the underlying cost function in bipartite graph matching to approximate the graph edit distance, an improved trade-off between running time and approximation accuracy can be achieved. To gain a more fine-grained vertex similarity measure, we propose a more gradual version of the Weisfeiler-Leman algorithm that converges to the stable coloring more slowly. The resulting similarity measure significantly improves the accuracy of graph edit distance approximations and graph kernels. Message passing neural networks can be seen as a neural version of the Weisfeiler-Leman algorithm. We develop message passing on pruned unfolding trees that represent node neighborhoods without information redundancy. This type of redundancy has been linked to oversquashing. We mitigate computational redundancy, by merging isomorphic substructures in the computational graph. Therefore, we address both types of redundancy present in graph neural networks.

Furthermore, we propose a linear-time graph transformation that enables the Weisfeiler-Leman algorithm to distinguish any two non-isomorphic outerplanar graphs. This is especially relevant for molecular data, as most molecules can be represented as outerplanar graphs. The expressivity of message passing neural networks in relation to the Weisfeiler-Leman algorithm has been studied extensively, but expressivity on specific graph classes has not yet been widely investigated.

In this thesis we advance graph learning by finding a better trade-off between efficiency and accuracy in approximating the graph edit distance, specifically focusing on accurate representations of node neighborhoods. Furthermore, we apply our representations to graph neural networks to develop more expressive models.

Kurzfassung

Graphen dienen als vielseitige Darstellungen für reale und auch abstrakte Objekte in vielen Anwendungsbereichen, zum Beispiel in der Analyse von sozialen Netzwerken, dem Gesundheitswesen und bei der Erkennung von betrügerischen Aktivitäten. Aufgaben im Graphlernen, die von der Klassifikation von Knoten und der Vorhersage von Kanten, bis hin zur Klassifikation von Graphen reichen, werden immer häufiger. Mit zunehmender Menge an Daten, ist die Skalierbarkeit von Graphlernmethoden entscheidend geworden. Außerdem stellt die Expressivität, die die Fähigkeit einer Methode misst, Paare von nicht-isomorphen Graphen zu unterscheiden, immer noch eine grundlegende Herausforderung im Graphlernen dar.

Ziel dieser Arbeit ist es, Graphlernverfahren voranzutreiben, indem die Skalierbarkeit erhöht, Ähnlichkeitsmaße verbessert und die Beschränkungen von derzeitigen Methoden in Bezug auf die Expressivität untersucht werden. Da Ähnlichkeitsmaße für Graphen typischerweise auf Ähnlichkeitsmaßen für Knoten beruhen, untersuchen und entwickeln wir Methoden, um die Nachbarschaft von Knoten genauer zu erfassen. Insbesondere konzentrieren wir uns auf die Approximation der Grapheditierdistanz zur effizienten Suche in Graphdatenbanken. Durch die Verwendung einer bestehenden unteren Schranke für die Grapheditierdistanz in Kombination mit einem metrischen Index in einem Filter-Verifikations-Framework ermöglichen wir die Suche in großen Graphdatenbanken. Wir schlagen neue untere Schranken vor, die in den ℓ_1-Raum eingebettet werden können, um als noch effizientere erste Filter zu dienen. Wir entwickeln kompakte Bäume, die Knotennachbarschaften ohne Redundanz kodieren. Durch die Verwendung der Distanz dieser Bäume als zugrundeliegende Kostenfunktion beim *Bipartite Graph Matching* zur Approximation der Grapheditierdistanz, kann ein besserer Kompromiss zwischen Laufzeit und Approximationsgüte erreicht werden. Um eine feingliedrigere Knotenähnlichkeit zu erhalten, entwerfen wir eine graduellere Version des Weisfeiler-Leman-Algorithmus, die langsamer zur stabilen Färbung konvergiert. Die sich daraus ergebende Ähnlichkeitsfunktion verbessert die Genauigkeit bei der Approximation der Grapheditierdistanz und in Graphkernen erheblich.

Das *Message Passing* in Graphneuronalennetzen kann als neuronale Version des Weisfeiler-Leman-Algorithmus gesehen werden. Wir entwickeln Message Passing auf kompakten Bäume, die Knotennachbarschaften ohne Informationsredundanz darstellen. Diese Art von Redundanz wurde mit *Oversquashing* in Verbindung gebracht. Außerdem verringern wir die Berechnungsredundanz, indem wir isomorphe Teilstrukturen im Berechnungsgraphen zusammenführen. Wir gehen also auf beide Arten von Redundanz ein, die in Graphneuronalennetzen vorkommen.

Zusätzlich entwickeln wir eine linear-zeit Graphtransformation, die es dem Weisfeiler-Leman-Algorithmus ermöglicht, zwei beliebige nicht-isomorphe außerplanare Graphen zu

unterscheiden. Dies ist besonders wichtig für Molekülдaten, da die meisten Moleküle als außerplanare Graphen dargestellt werden können. Die Expressivität von Graphneuronalennetzen in Bezug auf den Weisfeiler-Leman-Algorithmus wurde ausgiebig untersucht, aber die Expressivität für bestimmte Graphklassen wurde noch nicht umfassend erforscht.
In dieser Arbeit entwickeln wir Graphlernverfahren weiter, indem wir einen besseren Kompromiss zwischen Effizienz und Genauigkeit bei der Approximation der Grapheditierdistanz finden, wobei wir uns besonders auf die genauere Darstellung von Knotennachbarschaften konzentrieren. Außerdem wenden wir unsere Erkenntnisse auf Graphneuronalenetze an, um expressivere Modelle zu entwickeln.

Contents

1 Introduction

Graphs are everywhere, and graph learning has become increasingly prevalent in various fields. Graphs consisting of nodes and edges represent both real-world objects, such as molecules, and abstract structures, such as social networks. Graph learning includes a variety of tasks within graphs, such as node classification and link prediction, as well as graph-level tasks, such as graph classification and regression. In social networks, tasks such as community detection and user-specific recommendations can be solved, while in healthcare, drug discovery plays an important role [GFS19]. In transportation systems, graph learning can predict traffic [ZZL+22] or optimize routes [LGW+22]. Fraudulent or abnormal transactions can be detected within transaction networks [MR24]. In addition, graph-based models have also been used in computer vision for object detection and scene understanding [BCH12].

In recent years, the amount of data that has to be processed has steadily increased in nearly all applications. Therefore, a method must scale to large datasets. Index structures are specialized data structures that can help search or retrieve data efficiently. Therefore, methods that utilize index structures can be very efficient when handling large amounts of data. Graph learning methods must also scale to large datasets to be useful in practice [YHC+18].

Graphs are complex data structures, and deciding whether two graphs are the same, meaning they are isomorphic, is already a challenging problem. Currently, no polynomial time algorithm for the graph isomorphism problem is known [GS20], and most methods used in practice cannot distinguish all pairs of non-isomorphic graphs. Any method that cannot distinguish two non-isomorphic graphs will not be able to compute different values or representations for them. Therefore, a method's ability to distinguish non-isomorphic graphs, called expressivity, also plays an important role in graph learning.

However, solving graph-related tasks requires more than just the binary decision of whether two graphs are the same or not; it requires a notion of similarity. For instance, in chemistry, it is often assumed that similar molecules have similar properties. When analyzing user behavior, we might expect the interaction patterns of real users to differ significantly from those of bots.

However, comparing graphs is not straightforward. Determining whether two graphs are more similar to each other than two other graphs can be challenging, and in some cases, it is even difficult to define what similarity means. This similarity could be based more on the structural properties of the graph or on its features (but often, we expect it to be a mix of both).

Graph learning methods commonly rely on one of the following concepts for graph similarity: graph edit distance, graph kernels, and graph neural networks. All three concepts typically define graph similarity based on the similarity or embeddings of

individual nodes. To achieve this, they consider the neighborhood structure around each node, capturing both local and global patterns in the graph.

The graph edit distance is an intuitive and interpretable method for comparing graphs, which quantifies the cost of transforming one graph into the other. However, its exact computation is $\mathcal{NP}$-hard [ZTW+09] and not feasible for larger graphs, making it impractical for many real-world applications. To address this, many approximations for the graph edit distance have been proposed, several of them use so-called bipartite graph matching [RB09]. This technique computes an upper bound for the graph edit distance by finding an optimal assignment between the nodes of the two graphs and deriving a suboptimal edit path from this assignment. Other approaches, often more accurate than bipartite graph matching, use approximate versions of exact algorithms, such as relaxed binary linear programming or beam search techniques [BBG+20]. While these methods sometimes outperform others regarding accuracy, their computational complexity still renders them impractical for large-scale use.

Graph kernels are a traditional and widely used similarity measure for graphs. Although they often rely on hand-crafted features, they have achieved good accuracy in practice. Many graph kernels work by comparing the counts of specific substructures within the graphs. A particularly popular family of graph kernels are the Weisfeiler-Lehman graph kernels [SSvL+11], which are based on the Weisfeiler-Leman algorithm. The Weisfeiler-Leman algorithm [GKMS21, WL68], a heuristic for the graph isomorphism problem, iteratively colors the nodes of the graphs based on their current color (initially set to the node label) and the multiset of colors of their neighbors. In each iteration, two nodes get the same color if and only if they had the same color previously, and the multisets of neighbor colors are equal. Although this approach provides a fast and straightforward way to derive a similarity function for nodes, this similarity is often not fine-grained enough. In order to develop better similarity measures for graphs, a suitable similarity measure for the nodes is crucial.

Recently, graph neural networks (GNNs) have emerged as a popular tool for tackling graph learning tasks. In GNNs, typically, embeddings for the nodes or graphs are learned (in some cases, also embeddings for edges). In message passing GNNs, nodes communicate with their neighbors by sending messages. The embedding of a node is updated by combining its previous embedding with an aggregation of the messages received from its neighbors. This process is conceptually very similar to the Weisfeiler-Leman algorithm. It has been shown that if the Weisfeiler-Leman algorithm fails to distinguish two graphs, message passing GNNs will also fail, computing identical embeddings for them [MRF+19]. This limitation highlights the importance of expressivity, the ability to differentiate non-isomorphic graphs. While GNNs often achieve superior prediction accuracy compared to the graph edit distance and graph kernels, their expressivity and generalization capabilities remain areas of ongoing research. Furthermore, their results are often difficult to interpret, and training GNNs requires substantial amounts of data and computational resources.

Conceptually, the graph edit distance provides a tangible distance measure, graph kernels offer a middle ground by using predefined features to compute a similarity measure,

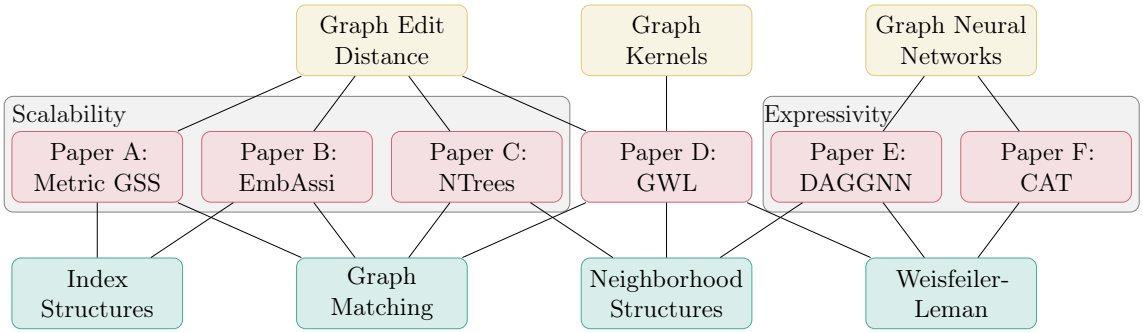


Figure 1.1: Overview of the papers (red) in this thesis, their main method for graph learning used (yellow), and main areas of the contributions in them (teal). Approximating the graph edit distance efficiently is the most prominent topic in papers A to C, while paper D focuses on the Weisfeiler-Leman algorithm. In papers E and F the focus is set on graph neural networks and expressivity.

and GNNs provide representations based on the task and data at hand, allowing them to handle complex graph-based tasks.

1.1 Overview of Publications

In general, our research focuses on developing efficient and expressive methods for graph learning, including the design of scalable similarity measures for graphs and more expressive graph learning methods. We aim to improve both the accuracy and running time of existing methods and discover new approaches. Central to our research is the Weisfeiler-Leman algorithm, and since graph similarity is usually defined through node similarity, how to capture a node's neighborhood accurately.

Figure 1.1 provides an overview of how the different contributions are structured with respect to their main topics and methods for graph learning. Searching a database for graphs similar to a query graph is a common task. The increasing amount of data in recent years makes this task even more challenging. Therefore, finding ways to efficiently search in graph databases and improve the trade-off between running time and accuracy of approximations for the graph edit distance is important. In **Paper A:** *Metric Indexing for Graph Similarity Search* [BBSK21], we use an existing lower bound for the graph edit distance to accelerate search in large graph databases, while in **Paper B:** *EmbAssi: Embedding Assignment Costs for Similarity Search in Large Graph Databases* [BSK22], we develop new lower bounds for the graph edit distance, that can be used as an even more efficient pre-filter for graph database search. Also focusing on approximating the graph edit distance, in **Paper C:** *Approximating the Graph Edit Distance with Compact Neighborhood Representations* [BPK24], we propose k-redundant neighborhood trees and use them in the bipartite graph matching framework. These trees provide an accurate and compact representation of a node's neighborhood, and the distance between them can be used to find a better edit path via optimal assignments.

In **Paper D:** *Gradual Weisfeiler-Leman: Slow and Steady Wins the Race* [BK22], we modify the Weisfeiler-Leman algorithm by slowing down its convergence, resulting in a more fine-grained node similarity. We use this approach both in graph kernels as well as for approximating the graph edit distance by swapping the original Weisfeiler-Leman algorithm with ours in the respective methods. By developing a more fine-grained node similarity measure, we can capture graph similarity more effectively.

We use the k-redundant neighborhood trees proposed in **Paper C** to eliminate information redundancy in message passing GNNs in **Paper E:** *On the Two Sides of Redundancy in Graph Neural Networks* [BML+24]. We also investigate computational redundancy and propose ways to mitigate it. Information redundancy has been identified as a contributor to so-called oversquashing. Therefore, reducing redundancy helps to increase performance in practice.

In **Paper F:** *Maximally Expressive GNNs for Outerplanar Graphs* [BJI+25], we develop a linear time graph transformation that allows the Weisfeiler-Leman algorithm and sufficiently expressive message passing GNNs to distinguish any two non-isomorphic outerplanar graphs. The class of outerplanar graphs is especially interesting since most molecules can be represented as outerplanar graphs. Developing a method to make these graphs distinguishable from each other without substantial computational overhead is a step toward more expressive models for specific graph classes.

Together, these contributions aim to advance the efficiency and expressivity of graph learning by refining node similarity measures and by utilizing and extending the Weisfeiler-Leman algorithm across various graph learning methods.

1.2 Thesis Structure

This thesis is structured as follows: In Chapter 2, the necessary background on graph theory and graph learning is provided. Chapter 3 highlights the key contributions, which are further examined in the conclusion (Chapter 4) with respect to their scientific relevance and potential future directions. The papers underlying these contributions, along with additional details such as publication venues and author contributions, are presented in Appendix A.

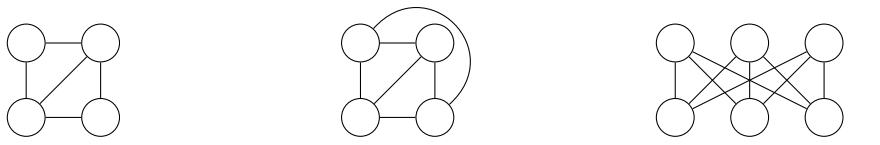
2 Background

This chapter provides key theoretical definitions to support the contributions presented in this thesis. Each relevant topic is discussed briefly and only to the necessary extent, as the papers in Appendix A are self-contained. The chapter begins with fundamental concepts of graph theory, followed by graph learning and its associated methods, and concludes with an overview of database search.

2.1 Graph Theory

A *graph* consists of nodes, or vertices, that are connected by edges. A graph can be directed or undirected. In undirected graphs, an edge connects two vertices in both directions. In directed graphs, every edge has a start and an end vertex. Vertices connected by an edge are *neighbors* of each other. There is at most one edge between two vertices in graphs and possibly more than one edge in multigraphs. Vertices and edges can have features, that are usually specific to the application or dataset. They can be categorical features, such as the atom type of vertices in a molecular graph, continuous attributes, for example, in the case of coordinates, or even a mix of both.

An undirected graph is *connected*, if any vertex can be reached by any other vertex following the edges, and a connected graph is *biconnected*, if it has at least three vertices, and removing any vertex yields a connected graph. A graph is *planar*, if it can be drawn in the plane, without any edges crossing. Similarly, a graph is *outerplanar*, if it can be drawn in the plane, without any edges crossing and so that all vertices touch the surrounding space. An outerplanar graph, a graph that is planar but not outerplanar, and a graph that is not planar are shown in Figure 2.1 (see [Fel04] for further details on planar graphs). A *Hamiltonian cycle* is a cycle which contains each node of the graph exactly once, and biconnected outerplanar graphs have a unique Hamiltonian cycle [Mit79].



(a) Outerplanar graph (b) Planar graph (c) Non-planar graph

Figure 2.1: Graph (a) is outerplanar, because all vertices touch the surrounding space, while graph (b) is not outerplanar. Graph (c) is not planar, since it cannot be drawn without edges crossing.

(a) G (b) H

Figure 2.2: Two undirected graphs. The colors represent vertex labels.

Figure 2.2 shows two graphs that will be used to illustrate concepts and methods in this chapter.

An undirected *tree* is a connected graph without any cycles, which means that for each vertex, there is exactly one way to reach any other vertex via edges. In a rooted tree, one vertex is called the *root* of the tree. In rooted trees, we call the neighbors of a vertex further away from the root the *children* of this vertex, and the neighbor closer to the root the *parent*. This way, the root is the only vertex with no parent. Vertices without children are called *leaves*.

Two graphs are *isomorphic*, if there exists a bijective mapping between their vertex sets, so that edges and features are preserved. Deciding whether two graphs are isomorphic is a difficult problem (and it is still not known, whether it is $\mathcal{NP}$-complete or can be solved in polynomial time [GS20]). The ability of a function to distinguish two non-isomorphic graphs from each other is called its *expressivity*. One method is more expressive than the other if it can distinguish (strictly) more pairs of non-isomorphic graphs, including all graphs that the other method can distinguish. A method is *maximally expressive* on a set of graphs if it can distinguish all pairs of non-isomorphic graphs in that set.

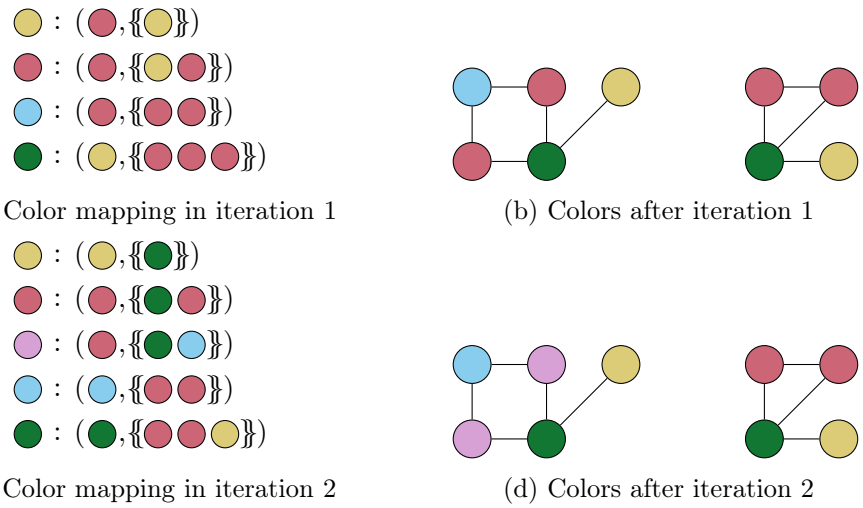
Weisfeiler-Leman and Unfolding Trees. A popular method, which is very prominent in this thesis, is the *Weisfeiler-Leman*[1] (WL) or *color refinement* [WL68, GKMS21] *algorithm*, which was originally proposed to solve graph isomorphism. It will produce the same result for any two isomorphic graphs. However, there are non-isomorphic graphs, that it cannot distinguish, making it only a heuristic for graph isomorphism.

The Weisfeiler-Leman algorithm works by iteratively coloring the vertices of graphs according to their neighborhood. It starts with a coloring c_0, where all vertices have a color representing their label (or a uniform coloring in the case of unlabeled vertices). In iteration i, the coloring c_i is obtained by assigning each vertex v a new color according to its color from the previous iteration and the colors of its neighbors $N(v)$, i.e.,

$$c_{i+1}(v) = f\left(c_i(v), \{\!\{c_i(u) \mid u \in N(v)\}\!\}\right),$$

where f is an injective function. An example of the first two iterations of the Weisfeiler-Leman algorithm on the graphs from Figure 2.2 is given in Figure 2.3. A coloring allows to partition a set of vertices into subsets of vertices with the same color. The algorithm

[1] We use the spelling *Leman* over Lehman, as preferred by Andrew Leman (see also [MLM+23]). For methods including the h, such as the Weisfeiler-Lehman kernels [SSvL+11], we include it as well.



(a) Color mapping in iteration 1

(b) Colors after iteration 1

(c) Color mapping in iteration 2

(d) Colors after iteration 2

Figure 2.3: The first two iterations of the Weisfeiler-Leman algorithm for the two graphs from Figure 2.2. Colors are reused between iterations. The mapping to new colors is depicted in (a) and (c).

eventually reaches a so-called *stable coloring*, meaning that this partitioning does not change anymore.

If the multisets of vertex colors of two graphs differ in any iteration, they cannot be isomorphic. However, there are non-isomorphic pairs of graphs, that cannot be distinguished by the Weisfeiler-Leman algorithm; for example, the Weisfeiler-Leman algorithm cannot distinguish a 6-cycle from two triangles.

Colors generated by the Weisfeiler-Leman algorithm encode so-called unfolding trees, which can be used to represent the neighborhood of a node. If the unfolding trees with height h of two vertices differ, their colors c_h also differ, and vice versa.

Mathematically, the *unfolding tree* F_h^v with height h of the vertex v is defined recursively as the tree with a root n_v that represents v, and child subtrees F_{h-1}^w for all $w \in N(v)$, and F_0^v consisting only of the node n_v. The node and edge labels of the original graph are preserved. An example of the unfolding trees with height 2 of the vertices of graph H is given in Figure 2.4.

While graph isomorphism poses as a way to decide whether two graphs are equal, the similarity measure given by it is binary. Two graphs are either isomorphic or they are not. However, this is not sufficient in many applications, and a notion of similarity is needed. In drug discovery, for example, it would not make much sense to search for the same molecule; instead, the task is usually to search for similar molecules. Similarity is also important in graph learning, as it is typically assumed that similar graphs have similar properties.

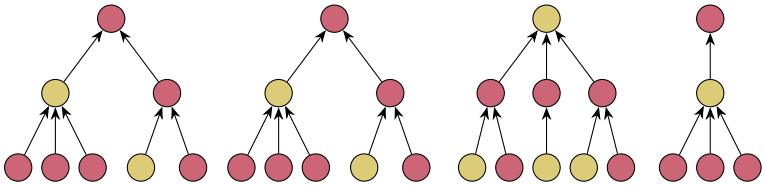


Figure 2.4: Unfolding trees with height 2 of the nodes of H.

2.2 Graph Learning

Graph learning refers to learning representations or making predictions for graph-structured data. This is often done in a supervised manner, where a training set with known labels is used to train a model. The model learns from this labeled data, and after optimizing its parameters, it is evaluated on a test set, which consists of data it has not encountered during training. If the model performs well on this test set, it is considered to generalize well.

Graph learning tasks can be categorized into graph-level, node-level, and edge-level tasks. A common graph-level task is graph classification, where the goal is assigning a class label to an entire graph, such as determining whether a molecule is toxic. Graph regression involves assigning a numerical value to a graph, such as predicting the solubility of a molecule. These tasks can also be applied at the node or edge level. At the edge level, another possible task is link prediction, which aims to predict whether an edge should exist between two nodes. This is particularly useful in applications such as social networks or recommender systems, where predicting potential connections between entities is essential.

To solve graph learning tasks, typically, a similarity measure for graphs is needed. Defining a similarity measure between graphs is not trivial and often depends on the graphs and the task. Since the applications of graphs are very diverse, many researchers have focused their attention on developing similarity measures suitable for graphs. In the following, different approaches for graph similarity and graph learning are described.

Graph Edit Distance

The *graph edit distance* (GED) is a general distance measure between graphs. It describes how expensive it is to transform one graph into the other using a sequence of edit operations (also called an edit path). These edit operations usually include inserting or deleting edges or isolated vertices, or changing the attribute of either of them. Edit operations have costs assigned to them, depending on the task or often uniform costs for simplicity reasons. Formally, the graph edit distance d_{GED} is defined as the cost of an optimal (minimum cost) edit path between the two graphs:

$$d_{\text{GED}}(G, H) = \min \left\{ \sum_{i=1}^{k} c(e_i) \;\middle|\; (e_1, ..., e_k) \in \Upsilon(G, H) \right\},$$

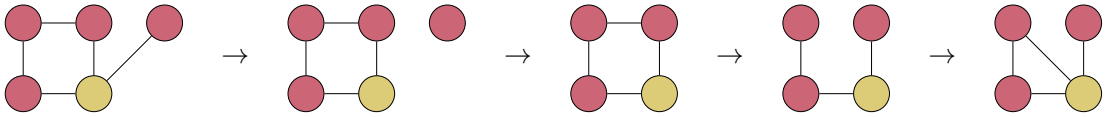


Figure 2.5: An edit path between the two graphs G and H from Figure 2.2. If the cost of each edit operation is defined as 1, this edit path has a cost of 4 and is optimal, meaning that $d_{\mathrm{GED}}(G, H) = 4$.

where $\Upsilon(G, H)$ is the set of all possible edit paths between G and H, and $c(e_i)$ is the cost of edit operation e_i. Figure 2.5 gives an example of an edit path.

Graph-level learning tasks, for example, can be solved by using the graph edit distance for k-nearest neighbor classification. The costs of the edit operations are typically tuned manually but have also been learned to adjust the graph edit distance to the task at hand [GHFS20].

Exact algorithms for computing the graph edit distance use integer linear programming [LAR+17] or are based on depth- or breadth-first search [CFL+20, GH16]. However, computing the graph edit distance (even with all edit costs defined as 1) is an $\mathcal{NP}$-hard problem [ZTW+09], and exact computation is often not feasible. Therefore, many approximations have been proposed, for example, [KGBW19, RB09, BBG+20, NRB06, BG18]. Some approximations can be derived from exact algorithms, for example, using linear programming relaxations or beam search [NRB06] instead of the A^* algorithm. Others are based on finding a suboptimal edit path, which is an upper bound for the exact distance.

Approximating the Graph Edit Distance. One method of finding a suboptimal edit path is to first find an (optimal) assignment between the vertices of the graphs. The assignment is optimal in that it minimizes the costs regarding an underlying cost function. The assignment problem can be solved optimally in cubic time using the Hungarian algorithm [Mun57] or approximated in quadratic time using simple greedy strategies [RFFB15], and a suboptimal edit path can be derived from the optimal assignment to gain an upper bound for the graph edit distance. This type of heuristic is widely used and successful in practice [RB09, BG18].

However, defining the cost matrix for the optimal assignment problem is difficult. A popular approach uses the difference in local structures around the nodes as the cost function [RB09], while a very similar approach, called Branch [BG18], adjusts these costs carefully to ensure that the cost of the optimal assignment is a lower bound for the graph edit distance. This means that with this approach, a lower and an upper bound can be computed by solving only one assignment problem. While another benefit of this approach is that the lower bound computed this way is a metric, the runtime still is not feasible for processing large amounts of data.

In general, there is a trade-off between runtime and quality of approximation, and with lower quality of approximation, the expressivity might also decrease as the graph edit distance becomes a coarser distance measure, while the scalability usually increases.

Number of nodes with color in	Initial labels		Colors from iteration 1				Colors from iteration 2				
											
G	1	4	1	2	1	1	1	0	2	1	1
H	1	3	1	2	0	1	1	2	0	0	1

Figure 2.6: The Weisfeiler-Lehman subtree kernel WLST_2 between the two graphs G and H from Figure 2.2 can be computed from the inner product of their feature vectors, which consist of the number of nodes per color in each iteration. This is equivalent to the formulation given in Equation 2.1 [SSvL+11]. Following the definition, $\mathrm{WLST}_2(G, H) = 21$.

Methods that compute an optimal assignment between the vertices of the graphs or find a (possibly suboptimal) edit path are more easily interpreted due to the one-to-one correspondence of the vertices.

Graph Kernels

A *kernel* is a positive-semidefinite function that describes the similarity between two objects. In recent years, graph kernels have become a commonly used tool for graph learning. A wide variety of graph kernels have been proposed, and many were quite successful in solving graph classification tasks. Usually, these kernels are based on counting common substructures, such as graphlets [SVP+09], paths [BK05], walks [KTS12] or vertices with the same neighborhood [KM12].

The Weisfeiler-Lehman kernels [SSvL+11], which are based on the Weisfeiler-Leman algorithm, are very popular and commonly used. The Weisfeiler-Lehman subtree kernel with h iterations (WLST_h) is defined as the common occurrences of node colors up to iteration h, i.e.,

$$\mathrm{WLST}_h(G, H) = \sum_{i=0}^{h} \sum_{u \in V(G)} \sum_{v \in V(H)} \delta(c_i(u), c_i(v)), \tag{2.1}$$

where δ is the Dirac kernel (it is 1 if the two inputs are equal, and 0 otherwise). Figure 2.6 shows an example of the Weisfeiler-Lehman subtree kernel for the two graphs from Figure 2.2.

The Weisfeiler-Lehman graph kernels are only defined for graphs with discrete labels. For graphs with attributed vertices, other kernels have been proposed, for example, the Wasserstein Weisfeiler-Lehman graph kernels [TGL+19] or hash graph kernels [MKKM16].

The main drawbacks of graph kernels are that the ones used in practice cannot distinguish certain non-isomorphic graphs, and many, for example the Weisfeiler-Lehman kernels, cannot even predict specific graph properties like connectedness or planarity [KMRS18]. In contrast to the graph edit distance, the similarity is more abstract and cannot be interpreted as a mapping between the vertices. Additionally, the features used in the kernels are limited and often predefined.

Graph Neural Networks

Graph neural networks (GNNs) have become popular for their ability to solve various graph learning tasks. Typically, graph neural networks learn an embedding for each node by iteratively aggregating information from its neighborhood. This can be seen as messages sent between nodes via the edges of the graph, and it allows to capture not only individual node features but also more complex patterns within the graph. The representation of a graph is usually computed using the representations of its nodes.

Message passing neural networks such as GIN [XHLJ19] can be seen as a neural variant of the Weisfeiler-Leman algorithm. The embedding of a vertex v in layer i is defined as

$$x_v^i = \text{COMBINE}\left(x_v^{i-1}, \text{AGGREGATE}\left(\{\!\{x_u^{i-1} \mid u \in N(v)\}\!\}\right)\right),$$

where the initial embedding x_v^0 is usually acquired by applying a multi-layer perceptron (MLP) to the vertex features, and AGGREGATE and COMBINE are realized by suitable functions with learnable parameters. For graph-level tasks, the embedding x_G^i of a graph G in layer i is defined as

$$x_G^i = \text{READOUT}\left(\{\!\{x_v^i \mid v \in V(G)\}\!\}\right),$$

where READOUT is a suitable function with learnable parameters.

In graph neural networks, the features that are important for the task are learned and not provided manually, which makes graph neural networks suitable for different tasks with little expert knowledge required. Due to their nature, graph neural networks are highly adaptable and can be applied to all graph learning tasks and datasets. But it has also been shown that the expressivity of message passing graph neural networks is limited: for distinguishing non-isomorphic graphs, message passing neural networks are at most as powerful as the Weisfeiler-Leman algorithm [MRF+19].

A graph neural network is essentially a black box, which leads to poor interpretability. Many methods have been proposed to combat this, trying to explain graph neural networks [YYGJ23]. Additionally, although graph neural networks can process large amounts of data, the training procedure is often time-consuming.

With an increasing number of layers, more and more information is aggregated into the embedding of each node. However, since the embeddings are of a fixed size, they might not accurately represent these growing node neighborhoods. This problem, referred to as *oversquashing* [AY21], makes it difficult for graph neural networks to learn the interaction of distant nodes.

2.3 Database Search

Databases can be used to efficiently store and find data. Therefore, searching in databases is a very common task. In this thesis, we focus on two types of queries when searching in

a database: range queries and k-nearest neighbor queries. A *range query* finds all objects with a distance no more than the specified range to the query object (see Figure 2.7).

If the distance is expensive to compute, using a so-called filter-verification principle makes sense. In this approach, different lower and upper bounds filter out a large portion of the database (in the best case). A function d' is a *lower bound* for another function d, if $d'(x, y) \leq d(x, y)$, and an *upper bound*, if $d'(x, y) \geq d(x, y)$ holds for all inputs x and y. Objects with a lower bound above the specified range can be dismissed since the exact distance will be even larger. Objects with an upper bound within the specified range can be added to the result without further verification. Only the remaining objects need to be verified by computing the exact distance.

A *k-nearest neighbor query* returns the k nearest objects to the query object (see Figure 2.8). If multiple objects are tied for being the kth nearest neighbor, usually all of them are returned, meaning the k-nearest neighbor query might return more than k objects in those cases.

Since the distance to the kth nearest neighbor is not known at the start of the query, the filter pipeline cannot be as clearly separated into filtering and verification as for range queries. However, the *optimal multi-step k-nearest neighbor search* algorithm [SK98] can be used: The database objects are retrieved from the database in ascending order according to a lower bound (regarding their distance to the query object). For each object retrieved, the exact distance is computed, and only the k objects with the smallest exact distance are maintained. The current kth smallest exact distance can be used as a bound for further searching: The search can be terminated once we have found at least k objects with an exact distance smaller than the lower bound of all remaining objects. This procedure is optimal in the sense that none of the exact distance computations could have been avoided [SK98].

Approaches for Graph Similarity Search. Most approaches relying on pairwise computation of the graph edit distance are not suitable for similarity search and, thus, may not be able to handle large amounts of data.

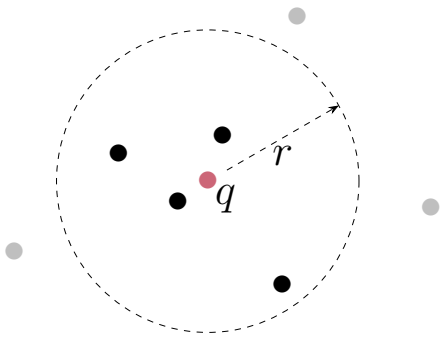


Figure 2.7: An example of a range query: All objects in the specified range r around the query object q are returned.

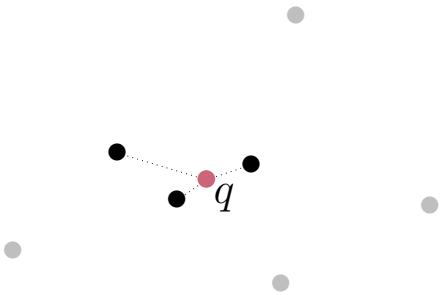


Figure 2.8: An example of a k-nearest neighbor query with $k = 3$. The 3 objects with the smallest distance to the query object q are returned.

Methods specifically proposed for similarity search in graph databases can mainly be divided into two categories, depending on whether they compare overlapping [WWYY12, ZTW+09, WDT+12, ZXLW12] or non-overlapping substructures [ZXL+13, LZ17, KCL19]. The methods that belong to the first category are inspired by q-grams used for computing the string edit distance and use either q-grams based on trees [WWYY12, ZTW+09, WDT+12] or paths [ZXLW12].

Methods that partition the graphs into non-overlapping substructures obtain lower bounds based on the observation that if x non-overlapping substructures of a database graph are not contained in the query graph, the graph edit distance is at least x. *Mixed* [ZZL+15] combines the idea of q-grams and graph partitioning.

These methods, however, only allow uniform edit costs (and some additionally cannot handle edge labels) and are, therefore, not suitable for graphs with continuous attributes or custom edit cost functions. Additionally, the bounds given by these approaches are often not tight enough to perform efficient similarity search on very large datasets.

3 Contributions

In the following sections, a brief overview of each paper and the key contributions made within them is presented. We explain each method briefly by using a figure describing the approach. Further details are presented in the corresponding papers, which are included in Appendix A.

3.1 Metric Indexing for Graph Similarity Search

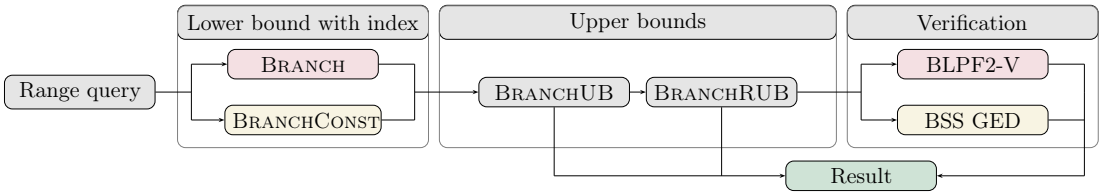


Figure 3.1: This figure shows the filter-verification pipeline of our approach. For a range query, first candidates are generated using the Branch lower bound together with an index structure. The candidates are then further filtered using the BranchUB upper bound and its refined version, BranchRUB. At these steps, results can already be reported. Only graphs, that could not be filtered out or be reported as results have to be processed using an exact algorithm in the verification step. For k-nearest neighbor queries, the optimal multi-step k-nearest neighbor search algorithm is used. The lower bound and exact algorithm used depend on whether the edit costs are uniform (yellow) or not (red). Figure adapted from [BBSK21].

In the paper *Metric Indexing for Graph Similarity Search* [BBSK21] (Appendix A.1), we develop a filter-verification framework for graph similarity search that is suitable for graphs with arbitrary attributes and non-uniform edit costs. This framework is depicted in Figure 3.1.

We employ the general, assignment-based lower bound Branch [BG18] together with a metric index to generate initial candidates. Branch approximates the graph edit distance by finding a minimal cost assignment between the vertices of the two graphs. The cost function is based on the cost of the edit operation and ensures that the cost of the optimal assignment is a lower bound of the graph edit distance. Since the Branch lower bound is a (pseudo-)metric for metric edit costs, we accelerated the candidate generation step using metric indexing. The generated candidates are further filtered using the upper bound BranchUB that is computed by deriving an edit path from the vertex assignment. If

needed, this edit path is subsequently refined in a local search (BranchRUB) by swapping the assignment of two vertices to generate a cheaper edit path. The remaining candidates are verified using one of two efficient verification algorithms. *BSS GED* [CHHV19] is very fast but only suitable for discrete labels and uniform edit costs, while *BLPF2-V* is used in the other cases.

Contribution. We develop a framework for fast graph similarity search using a lower bound for the graph edit distance, known to be a metric, along with a metric index structure and several upper bounds. In contrast to competing methods, our approach is general regarding dataset properties and edit costs, and is suitable for k-nearest neighbor search. Other approaches for similarity search in graph databases are typically restricted to uniform edit costs and can only answer range queries up to a threshold specified beforehand. In this work, we show that using the lower bound Branch together with a metric index in a filter-verification framework makes efficient similarity search possible, even when using attributed graphs and non-uniform edit costs. Specifically, range queries on a dataset consisting of over 100.000 graphs with attributes can be answered efficiently using our framework. Our approach outperforms competing methods, which can only handle graphs with discrete labels, on several datasets regarding filter quality and total query time.

3.2 EmbAssi: Embedding Assignment Costs for Similarity Search in Large Graph Databases

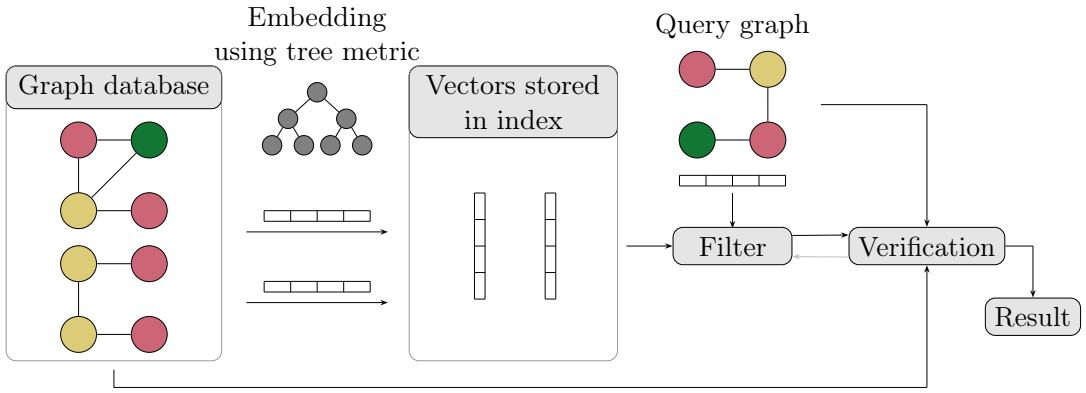


Figure 3.2: An overview of the filter-verification pipeline for graph similarity search used in our approach. First, the graphs of the database are embedded into vector space using a tree metric. These vectors are then stored in a suitable index structure. Any given query graph is also embedded and its resulting vector representation is used as the query in the first filtering step. Further filters can be applied before verification. Figure adapted from [BSK22].

In *EmbAssi: Embedding Assignment Costs for Similarity Search in Large Graph Databases* [BSK22] (Appendix A.2), we develop several lower bounds for the graph edit distance, that can be represented as the Manhattan distance between two vectors, making them suitable for usage with index structures for vectors. The lower bounds are derived from an assignment problem, where the cost function is a tree metric[1]. This allows embedding the costs of optimal assignments into ℓ_1 space [KGBW19]. The tree metrics that the lower bounds are based on are carefully constructed to reflect the edit costs, which do not have to be uniform. Figure 3.2 shows an overview of our framework.

Proposed Lower Bounds. We propose the label lower bound and the degree lower bound, which are based on the cost of an optimal assignment between the vertices of the graphs. The underlying cost functions for both bounds are tree metrics so that the cost of the optimal assignment can be represented as the Manhattan distance of two vectors. The two lower bounds are added in the combined lower bound to gain an even better approximation.

Conceptually, the *label lower bound* between two graphs G and H describes the minimal cost needed to change the node labels and the number of nodes in G, so that it matches those in H. The *degree lower bound* is based on the minimum number of edges that have

[1] A function $d : X \times X \to \mathbb{R}$ is a *tree metric* if there is a tree T with $X \subseteq T$ and strictly positive real-valued edge weights ω, so that $d(u, v) = d_{T,\omega}(u, v), \forall u, v \in X$ [KGBW19].

to be inserted and deleted so that the distributions of vertex degrees in the two graphs are equal.

The *combined lower bound* between G and H is the sum of the label and the degree lower bound, which can be represented by concatenating the vector representations computed for these two lower bounds.

Contribution. Using the vector representation of the combined lower bound together with an index (suitable for the Manhattan distance), we make efficient similarity search in large graph databases possible. We also integrate our approach into existing filter-verification pipelines as a fast pre-filtering step. Specifically, using our approach as the first filtering step, datasets of almost 2 million graphs can be processed.

Many methods for searching in graph databases rely on lower bounds based on non-overlapping substructures and indices specialized for graphs. They often can only answer range queries up to a maximum threshold, which has to be specified before index construction. Compared to competing methods for graph similarity search, our approach can answer both range and k-nearest neighbor queries and can be used with well-studied index structures for vector data. The proposed bounds can also be used with non-uniform edit costs, which is usually not possible for such approaches.

We relate our bounds to several commonly used lower bounds and show that they are very tight while being efficiently computable. We use our approach as a pre-filtering step in graph database search and show that the bounds generate manageable candidate sets quickly.

3.3 Approximating the Graph Edit Distance with Compact Neighborhood Representations

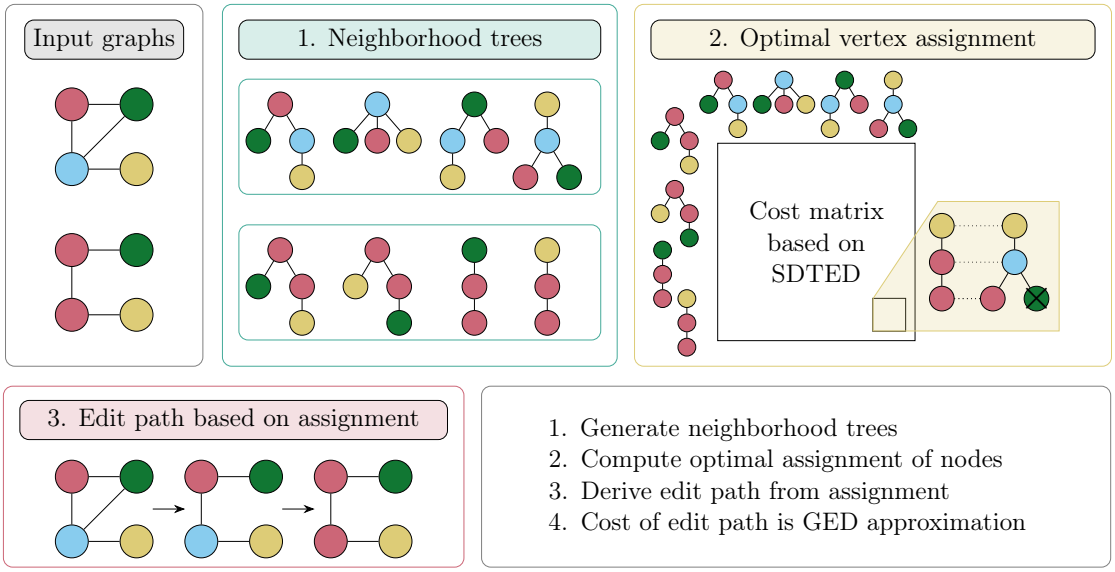
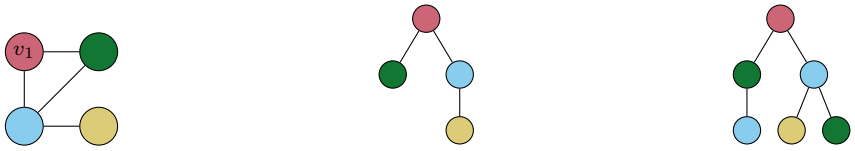


Figure 3.3: An overview of approximating the graph edit distance using our approach: For each vertex of the input graphs, a neighborhood tree is generated (step 1). In step 2 an optimal assignment between the vertices is computed. The cost of assigning a vertex to another is based on the structure-and-depth-preserving tree edit distance (SDTED) of their corresponding neighborhood trees. Once an assignment is found, an edit path can be derived from this assignment (by checking for each vertex and edge, whether any edit operation has to be performed on them to make them match their assigned counterpart) in step 3. The cost of this (possibly suboptimal) edit path is used as an upper bound for the graph edit distance (step 4). Figure adapted from [BPK24].

In *Approximating the Graph Edit Distance with Compact Neighborhood Representations* [BPK24] (Appendix A.3), we use trees representing the node neighborhood to approximate the graph edit distance in the bipartite graph matching framework. We propose k-redundant neighborhood trees, a compact representation of a node's neighborhood, that can be compared efficiently. Figure 3.3 shows an overview of our approach. First, a neighborhood tree is generated for each vertex. The distance between the neighborhood trees is used as the underlying cost function in the bipartite graph matching framework: an optimal assignment between the vertices of the two graphs is computed based on the cost function, and then the cost of an edit path derived from this assignment is used as the approximated graph edit distance.



(a) Graph (b) $T^{v_1}_{2,0}$ (c) $T^{v_1}_{2,1}$

Figure 3.4: A graph (a) and the 0-redundant and 1-redundant neighborhood tree with height 2, (b) and (c), respectively, of the vertex v_1.

k-redundant Neighborhood Trees. A k-redundant neighborhood tree (k-NT) $T^v_{h,k}$ of a node v with height h is derived from the unfolding tree F^v_h of v with height h by removing all subtrees with roots that occurred more than k levels before (seen from root to leaves). Figure 3.4 provides an example of 0- and 1-redundant neighborhood trees. More formally, let depth(v) denote the length of the path from v to the root, and $\phi(v)$ denote the original vertex in $V(G)$ represented by v in the unfolding or neighborhood tree, then $T^v_{h,k}$ is defined as the subtree of F^v_h, induced by its nodes u, that satisfy

$$\forall w \in V(F^v_h)\colon \phi(u) = \phi(w) \Rightarrow \text{depth}(u) \leq \text{depth}(w) + k.$$

Contribution. We propose k-redundant neighborhood trees as a compact way to represent node neighborhoods. We also improve the theoretical runtime for computing the structure-and-depth-preserving tree edit distance between two trees by conducting a thorough analysis. We use the structure-and-depth-preserving tree edit distance of neighborhood trees as the cost function in the bipartite graph matching framework. In contrast to competing approaches, our method yields a good trade-off between runtime and accuracy: While our method is slightly slower than standard bipartite graph matching, it achieves much lower approximation error. Experimentally, we showed that our method yields a better approximation, especially when the distance between the graphs is small, which is relevant for k-nearest neighbor classification. Compared to approaches using more complex neighborhood representations to compute the cost function for the optimal assignment, our method is multiple orders of magnitude faster and more accurate.

3.4 Gradual Weisfeiler-Leman: Slow and Steady Wins the Race

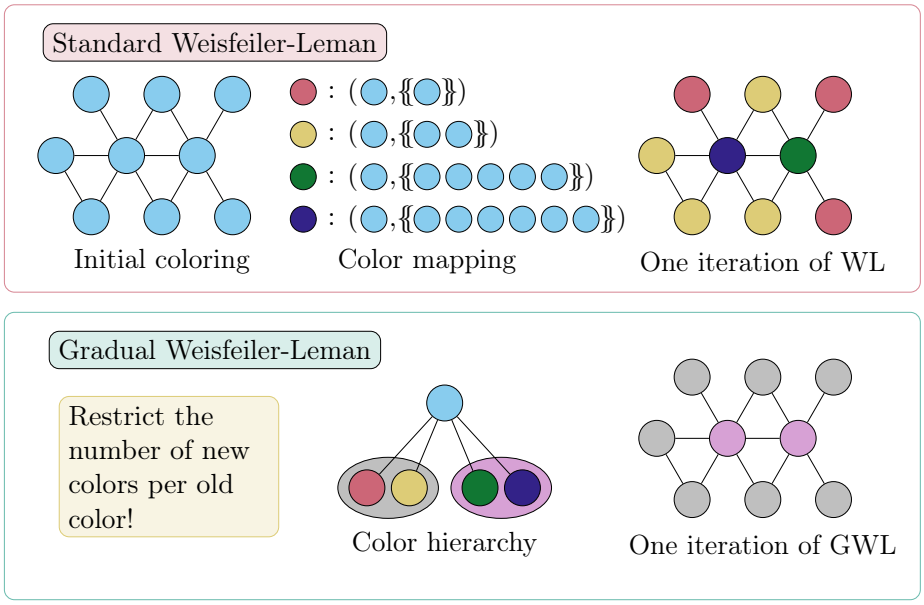


Figure 3.5: An overview of the gradual Weisfeiler-Leman refinement. The intuitive idea is to restrict the number of new colors in each iteration. The algorithm works similar to the original Weisfeiler-Leman algorithm, with the exception, that two nodes, which had the same color in the previous iteration, but have different multisets of neighbor colors now, might get assigned the same color again, given that at least two nodes that had the same color previously, get different colors. One can illustrate this using a color hierarchy, which describes how the colors evolve over the iterations. Using the gradual variant, colors of the original Weisfeiler-Leman might be merged into one color. Using, for example, k-means clustering to cluster the multisets of neighbor colors, the number of new colors can be restricted for each old color. Figure adapted from [BK22].

In *Gradual Weisfeiler-Leman: Slow and Steady Wins the Race* [BK22] (Appendix A.4), we propose a more gradual variant of the Weisfeiler-Leman algorithm. In the original Weisfeiler-Leman algorithm, two nodes get assigned the same color, if and only if they had the same color in the previous iteration and their neighbor color multisets do not differ. This makes sense for the original application of isomorphism testing, but the Weisfeiler-Leman algorithm has recently been used to define node similarities for multiple other applications. However, the similarity induced by the Weisfeiler-Leman color hierarchy, which describes how the colors evolve over the iterations, is not suitable for reflecting

minor differences. It cannot quantify the similarity of multiple nodes that get different colors in the same iteration.

For example, after the first iteration, nodes with differing degrees are assigned different colors. This means that a node with degree 5 will be just as similar to a node with degree 4 as a node with degree 100.

We tackle this problem by allowing nodes to get assigned the same color, even if their neighbor color multisets differ, given that overall, color classes are split up (see Figure 3.5). This allows for a more nuanced similarity function. We define the theoretical framework of *renep* functions as all functions that fulfill specific properties so that they are guaranteed to lead to the same stable coloring as the original Weisfeiler-Leman algorithm and use k-means clustering of the neighbor color multisets as a suitable and easy way to generate such a function.

Contribution. We propose a gradual version of the Weisfeiler-Leman algorithm to gain a more fine-grained node similarity by slowing down the convergence of the refinement process. We define which conditions must be fulfilled to obtain the same stable coloring as the original Weisfeiler-Leman algorithm. We show that the kernels derived from the gradual refinement surpass the original ones in accuracy, especially on social network data, and outperform all other kernels in 9 out of 12 datasets while being close to the best method in the other cases. Also, we use the gradual version to define a tree metric that can be used to find an optimal assignment between the vertices of two graphs for graph edit distance approximation and show that it is in general better than the original version, leading to a significant increase in classification accuracy (over 15% in some cases). Our gradual Weisfeiler-Leman algorithm can be used as a potent variation of the original version with little computational overhead. Especially applications where a more fine-grained node similarity is needed, for example, in social network data, can benefit from using the gradual Weisfeiler-Leman algorithm.

3.5 On the Two Sides of Redundancy in Graph Neural Networks

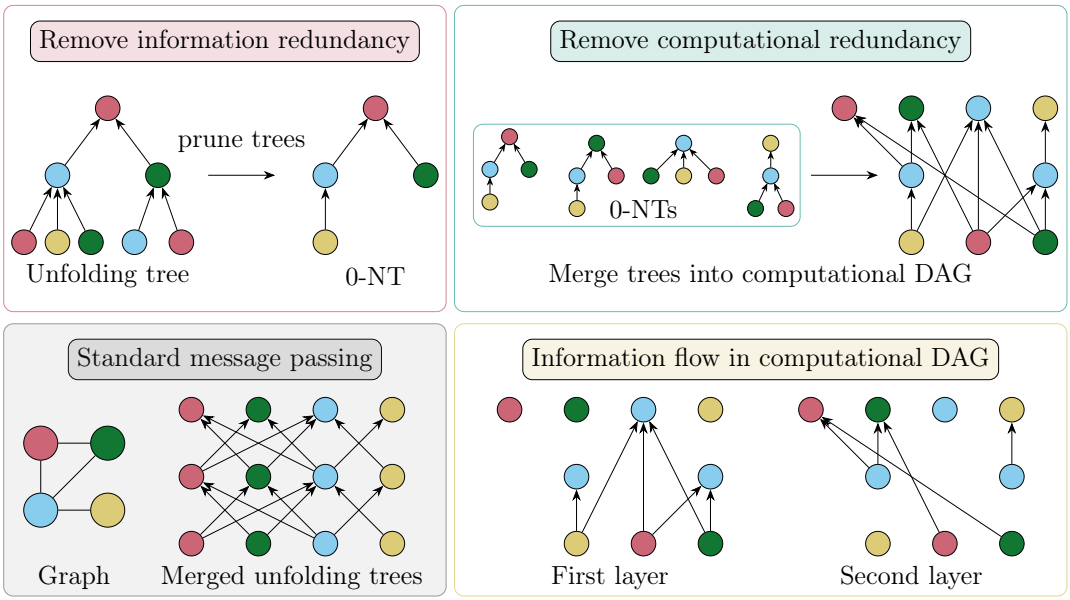


Figure 3.6: Overview of the DAGGNN framework. We investigate both information redundancy and computational redundancy in graph neural networks. We first propose to prune unfolding trees representing the computational trees of message passing neural networks, so that they do not contain information redundancy. To combat computational redundancy, we propose to merge isomorphic computational trees (and subtrees) into directed acyclic graphs (DAGs), from which node embeddings can be computed in a bottom-up fashion (see Information flow in computational DAG).

In the paper *On the Two Sides of Redundancy in Graph Neural Networks* [BML+24] (Appendix A.5), we investigate information and computational redundancy in GNNs and propose ways how to mitigate them. We use the k-redundant neighborhood trees developed in [BPK24] (refer to Section 3.3) to reduce information redundancy, and tree compression techniques to tackle computational redundancy (see Figure 3.6). Our approach, *DAGGNN*, performs message passing on directed acyclic graphs stemming from this tree compression.

Information Redundancy. Information redundancy occurs when a node sends a message to another node and this node then incorporates this message in its message back to the first node. This information redundancy can be seen when looking at the computational trees in message passing GNNs, the Weisfeiler-Leman unfolding trees. We link this redundancy of information to the phenomenon of oversquashing [DGGB+23] - the node neighborhood

influencing the fixed-size embedding grows exponentially with each message passing step, making it harder for distant nodes to communicate. We propose using k-redundant neighborhood trees for the information flow to remove the redundant information in message passing.

Computational Redundancy. We also investigate the other side of redundancy in GNNs - computational redundancy. Computational redundancy means that for vertices with the same neighborhood, the same embedding will be computed and is essentially computed twice. Our approach avoids computational redundancy by merging duplicate subtrees in the computational trees, yielding a directed acyclic graph as the computational graph. The information flow in standard message passing GNNs can be expressed as simple matrix multiplication using the graph's adjacency matrix. However, this is not possible when using trees other than the unfolding trees. Therefore, we show how to propagate information using arbitrary computational DAGs. The representation of a node in the DAG is only updated once and as soon as all its children's representations have been updated (see *Information flow in computational DAG* in Figure 3.6).

Contribution. We investigate information and computational redundancy in message passing GNNs and propose ways to reduce both. By using k-redundant neighborhood trees (refer to Section 3.3) as the computational trees in our message passing, we combat information redundancy, and by merging isomorphic substructures in the computational graph we eliminate computational redundancy. We prove that k-redundant neighborhood trees are incomparable to unfolding trees in terms of expressive power on the node level and show that oversquashing can be mitigated using our method. Our GNN architecture outperforms competing methods, especially on node classification tasks with low homophily ratio, with an improvement over the best other method of up to 12%.

3.6 Maximally Expressive GNNs for Outerplanar Graphs

A large percentage of molecules can be represented as outerplanar graphs. However, standard message passing GNNs cannot distinguish specific pairs of non-isomorphic outerplanar graphs, making it impossible for them to compute different embeddings (and different predictions) for them. In *Maximally Expressive GNNs for Outerplanar Graphs* [BJI+25] (Appendix A.6), we develop a graph transformation that enables the Weisfeiler-Leman algorithm to distinguish all non-isomorphic outerplanar graphs. Due to the connection of the Weisfeiler-Leman algorithm and message passing GNNs, this graph transformation also enables GNNs, which achieve the same expressivity as the Weisfeiler-Leman algorithm, to distinguish those graphs in theory. We show that empirically, this leads to better performance on molecule datasets, which primarily consist of outerplanar graphs.

We build on results by Colbourn and Booth [CB81] to first transform biconnected outerplanar graphs to make them distinguishable by the Weisfeiler-Leman algorithm using our graph transformation CAT*. We then extend CAT* to CAT, which can distinguish all outerplanar graphs.

CAT* and CAT. CAT* transforms biconnected outerplanar graphs by augmenting two copies of the graph, directing their edges in a specific way and annotating the edges with the distance of their corresponding nodes using only edges of the directed Hamiltonian cycle. This way, the so-called Hamiltonian adjacency list (HAL) sequences are encoded in the Weisfeiler-Leman unfolding trees. For two biconnected outerplanar graphs, these sequences are the same or cyclic shifts of each other, if and only if they are isomorphic [CB81]. Figure 3.7 shows an overview of CAT* and how the HAL sequences are encoded in the unfolding trees of graphs transformed using CAT*. By transforming all biconnected outerplanar components of outerplanar graphs and adding additional vertices that help determine the orientation of these components in the original graph (see Figure 3.7), our transformation CAT can enable the Weisfeiler-Leman algorithm to distinguish all non-isomorphic outerplanar graphs.

Contribution. By enabling the Weisfeiler-Leman algorithm to distinguish all non-isomorphic outerplanar graphs, we also allow message passing GNNs to distinguish these graphs. As many molecules can be represented by outerplanar graphs, this can help achieve better results on many molecular tasks. Using CAT, we improve the performance of message passing neural networks, such as GIN and GCN, on 6 and 8 out of 10 datasets, respectively.

Additionally, until recently, it was largely unexplored which classes of graphs different architectures could distinguish, and expressivity was measured using the Weisfeiler-Leman hierarchy. We provide a method that is maximally expressive on the specific class of outerplanar graphs and show that maximal expressivity can be achieved in linear time for these graphs.

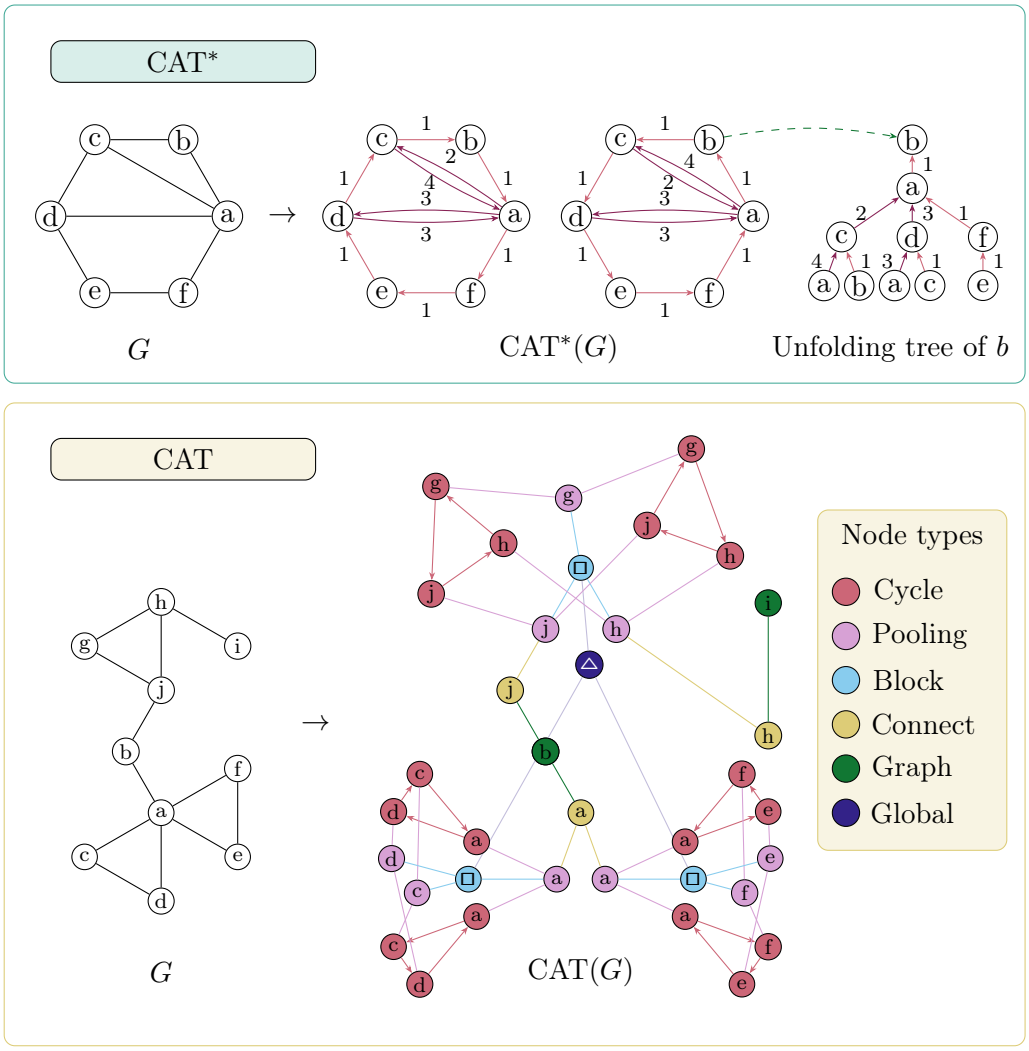


Figure 3.7: An overview of CAT* and CAT. In CAT* a biconnected outerplanar graph is transformed by copying the graph and directing the edges of its (unique) Hamiltonian cycle once in each direction. All other edges are present in both directions and are annotated with the distance of their endpoints on the Hamiltonian cycle. This encodes the Hamiltonian adjacency list sequence (which can be used to uniquely identify the graph) in the unfolding tree of each node. CAT takes any graph, applies CAT* to its biconnected outerplanar components, and adds further nodes to encode the position and orientation of these components within the graph. Letters represent original node labels, colors represent edge and node labels from CAT. *Cycle* nodes represent nodes added by CAT*, while *pooling* nodes are added to connect two corresponding cycle nodes. *Block* nodes connect the pooling nodes for each biconnected component, thereby able to store information about its HAL sequence, and a *global* node connects these nodes to reduce the diameter of the graph. *Connect* nodes connect the pooling nodes to the rest of the graph, thereby determining the orientation of the biconnected component. Using CAT, all outerplanar graphs can be distinguished by the Weisfeiler-Leman algorithm. Figure adapted from [BJI+25].

4 Conclusion

In this thesis, we developed both efficient and expressive graph learning methods, addressing two critical challenges. First, we introduced scalable similarity measures for graphs, enabling efficient comparison and search in large graph databases. Second, we developed graph learning methods with improved expressivity and little computational overhead. In the following, we briefly summarize the contributions of this thesis, outline potential directions for future research, and give some final remarks.

4.1 Contribution

Since the amount of data to be processed has drastically increased in recent years, scalability has become a significant challenge in graph learning. We proposed using the Branch lower bound for approximating the graph edit distance in combination with a metric index to enable search in large graph databases [BBSK21]. We showed that efficient search is possible, even when using attributed graphs and non-uniform edit costs. Our approach outperformed competing methods regarding filter quality and total query time, and additionally could efficiently handle attributed data of over 100.000 graphs.

We also developed lower bounds for the graph edit distance that can be embedded into ℓ_1 space [BSK22]. Using these lower bounds as an even more efficient first filter led to a significantly improved query time and made searching in large datasets of almost 2 million graphs possible.

Since graph similarity measures rely on similarity measures for nodes, we developed node similarities that can capture the neighborhood structure more accurately. We used the distance of pruned unfolding trees as the underlying cost function in bipartite graph matching to gain a significantly improved approximation with little increase in running time [BPK24], providing a better trade-off than competing methods. We slow down the convergence of the Weisfeiler-Leman algorithm by clustering similar nodes, which leads to a more fine-grained node similarity [BK22]. This gradual version of the Weisfeiler-Leman algorithm showed excellent performance when used in graph kernels, as well as for approximating the graph edit distance. Specifically, the proposed kernels outperformed competing approaches in 9 out of 12 datasets and were close to the best method in the other cases. When used to approximate the graph edit distance, our gradual version led to a significant increase in classification accuracy (over 15% in some cases) compared to the original Weisfeiler-Leman algorithm.

We also investigated and developed methods based on message passing neural networks, which are closely related to the Weisfeiler-Leman algorithm, with a special focus on expressivity. We used pruned unfolding trees [BPK24] to remove information redundancy

in message passing and showed how this can help mitigate oversquashing [BML+24]. We also proposed reducing computational redundancy by merging isomorphic substructures in the computational graph of message passing neural networks. Additionally, we proved that unfolding trees are incomparable to their pruned version regarding expressivity on the node level, and experiments indicated that they might have higher expressivity on the graph level. Our corresponding graph neural network architecture performs especially well on node classification tasks with a low homophily ratio, improving over the best other method by up to 12%.

We addressed one of the challenges (Challenge II.4: Towards expressiveness on relevant classes of graphs) brought up in a recent position paper [MFD+24] by focusing on the class of outerplanar graphs. Specifically, we developed a linear time graph transformation that enables the Weisfeiler-Leman algorithm to distinguish all pairs of non-isomorphic outerplanar graphs [BJI+25]. This is especially relevant for molecules, as most of them (up to 98% in common benchmark datasets [BJI+25]) can be represented as outerplanar graphs.

4.2 Limitations and Future Work

Our findings suggest several promising directions for future research. The notion of similarity for graphs could be considered in the context of the task at hand. While it is generally assumed that *structurally similar* graphs have similar properties, this is not always the case, for example, in molecules, where small structural changes can cause significant differences in properties (so-called activity cliffs [SHB19]). The graph edit distance is often used with fixed hand-picked edit costs, but learning a cost function for the specific task at hand as in [GHFS20] could improve its applicability. Therefore, learning meaningful distances for graphs beyond graph neural networks is an important challenge that should not be overlooked.

While methods related to graph matching, such as the graph edit distance, are usually inherently interpretable, they are not easily adapted to learning. However, graph embedding methods, such as graph neural networks, are highly adaptive, despite typically being a black box. Graph kernels provide a middle ground, as they can often adapt to the data at hand, but they still use predefined features and often require domain knowledge to achieve the best results. Combining these different underlying concepts could lead to both interpretable and adaptive models. Thus, future work can focus on bridging the gap between graph matching and graph embedding.

While this thesis does not focus on specific applications, we hope the findings can be helpful when applied to real-world problems. Given a specific application, the methods might need to be adapted slightly. Our findings already imply that, especially for social network data, the gradual Weisfeiler-Leman framework provides a better node similarity measure. Learning how to best partition nodes based on the task might yield an even better similarity measure, while also providing some interpretable insight into how different two nodes are from each other based on the task. Using the graph transformation developed in [BJI+25] in combination with the Weisfeiler-Lehman graph kernels might also be a possible future direction.

Our graph neural network architecture developed in [BML+24] performed especially well on node classification in heterophilic datasets. Therefore, the proposed method could be tailored more specifically to these datasets.

It might be worth further investigating the effect of information redundancy in graph neural networks and how it relates to expressivity. Developing models that are maximally expressive on other relevant graph classes and computationally feasible to use in practice could be another direction of future research.

The interplay between expressivity and the ability to generalize to new data is still an ongoing branch of research [MGTG23, FMVG24]. While methods that lack the expressivity to distinguish a pair of graphs cannot predict different values for them, it remains unclear why more expressive methods tend to generalize better. A different view on expressivity could be explored, for example, based on the ability to embed similar graphs to similar embeddings instead of solely distinguishing them.

4.3 Final Remarks

While the methods proposed in this thesis are only a small contribution in the fast evolving field of graph learning, we hope that these findings will inspire further research, especially into efficiently computable and more accurate similarity measures for nodes and graphs.

Bibliography

[AY21] Uri Alon and Eran Yahav. On the bottleneck of graph neural networks and its practical implications. In *9th International Conference on Learning Representations*. OpenReview.net, 2021.

[Bau20] Franka Bause. Efficient approximate k-nearest-neighbor-search in large graph databases. Master's thesis, TU Dortmund University, 2020.

[BBG+20] David B. Blumenthal, Nicolas Boria, Johann Gamper, Sébastien Bougleux, and Luc Brun. Comparing heuristics for graph edit distance computation. *The VLDB Journal*, 29:419–458, 2020.

[BBSK21] Franka Bause, David B. Blumenthal, Erich Schubert, and Nils M. Kriege. Metric indexing for graph similarity search. In *Similarity Search and Applications - 14th International Conference*, volume 13058 of *Lecture Notes in Computer Science*, pages 323–336. Springer, 2021.

[BCH12] Xiao Bai, J. Cheng, and Edwin Hancock. *Graph-Based Methods in Computer Vision: Developments and Applications*. Idea Group, 2012.

[BG18] David B. Blumenthal and Johann Gamper. Improved lower bounds for graph edit distance. *IEEE Transactions on Knowledge and Data Engineering*, 30(3):503–516, 2018.

[BJI+25] Franka Bause, Fabian Jogl, Patrick Indri, Tamara Drucks, David Penz, Nils M. Kriege, Thomas Gärtner, Pascal Welke, and Maximilian Thiessen. Maximally expressive GNNs for outerplanar graphs. *Transactions on Machine Learning Research*, 2025.

[BK05] Karsten M. Borgwardt and Hans-Peter Kriegel. Shortest-path kernels on graphs. In *Fifth IEEE International Conference on Data Mining*. IEEE, 2005.

[BK22] Franka Bause and Nils Morten Kriege. Gradual weisfeiler-leman: Slow and steady wins the race. In *Learning on Graphs Conference*, volume 198 of *Proceedings of Machine Learning Research*. PMLR, 2022.

[BML+24] Franka Bause, Samir Moustafa, Johannes Langguth, Wilfried N. Gansterer, and Nils M. Kriege. On the two sides of redundancy in graph neural networks. In *Machine Learning and Knowledge Discovery in Databases. Research Track*, volume 14946 of *Lecture Notes in Computer Science*, pages 371–388. Springer Nature Switzerland, 2024.

[BPK24] Franka Bause, Christian Permann, and Nils M. Kriege. Approximating the graph edit distance with compact neighborhood representations. In *Machine Learning and Knowledge Discovery in Databases. Research Track*, volume 14946 of *Lecture Notes in Computer Science*, pages 300–318. Springer Nature Switzerland, 2024.

[BSK22] Franka Bause, Erich Schubert, and Nils M. Kriege. Embassi: embedding assignment costs for similarity search in large graph databases. *Data Min. Knowl. Discov.*, 36(5):1728–1755, 2022.

[CB81] Charles J. Colbourn and Kellogg S. Booth. Linear time automorphism algorithms for trees, interval graphs, and planar graphs. *SIAM Journal on Computing*, 10(1):203–225, 1981.

[CFL+20] Lijun Chang, Xing Feng, Xuemin Lin, Lu Qin, Wenjie Zhang, and Dian Ouyang. Speeding up GED verification for graph similarity search. In *36th International Conference on Data Engineering*, pages 793–804. IEEE, 2020.

[CHHV19] Xiaoyang Chen, Hongwei Huo, Jun Huan, and Jeffrey Scott Vitter. An efficient algorithm for graph edit distance computation. *Knowledge-Based Systems*, 163:762 – 775, 2019.

[DGGB+23] Francesco Di Giovanni, Lorenzo Giusti, Federico Barbero, Giulia Luise, Pietro Liò, and Michael Bronstein. On over-squashing in message passing neural networks: the impact of width, depth, and topology. ICML'23. JMLR.org, 2023.

[Fel04] Stefan Felsner. *Geometric Graphs and Arrangements - Some Chapters from Combinatorial Geometry.* Advanced lectures in mathematics. Vieweg+Teubner, 2004.

[FMVG24] Billy Joe Franks, Christopher Morris, Ameya Velingker, and Floris Geerts. Weisfeiler-leman at the margin: When more expressivity matters. In *41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pages 13885–13926. JMLR.org, 2024.

[GFS19] Carlos Garcia-Hernandez, Alberto Fernández, and Francesc Serratosa. Ligand-based virtual screening using graph edit distance as molecular similarity measure. *Journal of Chemical Information and Modeling*, 59(4):1410–1421, 2019.

[GH16] Karam Gouda and Mosab Hassaan. CSI_GED: An efficient approach for graph edit similarity computation. In *32nd International Conference on Data Engineering*, pages 265–276. IEEE, 2016.

[GHFS20] Carlos Garcia-Hernandez, Alberto Fernández, and Francesc Serratosa. Learning the edit costs of the graph edit distance applied to ligand-based virtual screening. *Current Topics in Medicinal Chemistry*, 20(18):1582–1592, 2020.

[GKMS21] Martin Grohe, Kristian Kersting, Martin Mladenov, and Pascal Schweitzer. Color refinement and its applications. In *An Introduction to Lifted Probabilistic Inference*, pages 349–372. The MIT Press, 2021.

[GS20] Martin Grohe and Pascal Schweitzer. The graph isomorphism problem. *Communications of the ACM*, 63(11):128–134, 2020.

[KCL19] Jongik Kim, Dong-Hoon Choi, and Chen Li. Inves: Incremental partitioning-based verification for graph similarity search. In *Advances in Database Technology - 22nd International Conference on Extending Database Technology*, pages 229–240. OpenProceedings.org, 2019.

[KGBW19] Nils M. Kriege, Pierre-Louis Giscard, Franka Bause, and Richard C. Wilson. Computing optimal assignments in linear time for approximate graph matching. In *International Conference on Data Mining*, pages 349–358. IEEE, 2019.

[KM12] Nils M. Kriege and Petra Mutzel. Subgraph matching kernels for attributed graphs. In *Proceedings of the 29th International Coference on International Conference on Machine Learning*, page 291–298. Omnipress, 2012.

[KMRS18] Nils M. Kriege, Christopher Morris, Anja Rey, and Christian Sohler. A property testing framework for the theoretical expressivity of graph kernels. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence*, pages 2348–2354. AAAI Press, 2018.

[KTS12] U Kang, Hanghang Tong, and Jimeng Sun. Fast random walk graph kernel. In *Proceedings of the Twelfth SIAM International Conference on Data Mining*, pages 828–838. SIAM / Omnipress, 2012.

[LAR+17] Julien Lerouge, Zeina Abu-Aisheh, Romain Raveaux, Pierre Héroux, and Sébastien Adam. New binary linear programming formulation to compute the graph edit distance. *Pattern Recognition*, 72:254–265, 2017.

[LGW+22] Kun Lei, Peng Guo, Yi Wang, Xiao Wu, and Wenchao Zhao. Solve routing problems with a residual edge-graph attention neural network. *Neurocomputing*, 508:79–98, 2022.

[LZ17] Yongjiang Liang and Peixiang Zhao. Similarity search in graph databases: A multi-layered indexing approach. In *33rd International Conference on Data Engineering*, pages 783–794. IEEE, 2017.

[MFD+24] Christopher Morris, Fabrizio Frasca, Nadav Dym, Haggai Maron, Ismail Ilkan Ceylan, Ron Levie, Derek Lim, Michael M. Bronstein, Martin Grohe, and Stefanie Jegelka. Position: Future directions in the theory of graph machine learning. In *41st International Conference on Machine Learning*, volume 235 of *Proceedings of Machine Learning Research*, pages 36294–36307. JMLR.org, 2024.

[MGTG23] Christopher Morris, Floris Geerts, Jan Tönshoff, and Martin Grohe. WL meet VC. In *40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pages 25275–25302. JMLR.org, 2023.

[Mit79] Sandra L. Mitchell. Linear algorithms to recognize outerplanar and maximal outerplanar graphs. *Information Processing Letters*, 9(5):229–232, 1979.

[MKKM16] Christopher Morris, Nils M. Kriege, Kristian Kersting, and Petra Mutzel. Faster kernels for graphs with continuous attributes via hashing. In *16th International Conference on Data Mining*, pages 1095–1100. IEEE, 2016.

[MLM+23] Christopher Morris, Yaron Lipman, Haggai Maron, Bastian Rieck, Nils M. Kriege, Martin Grohe, Matthias Fey, and Karsten M. Borgwardt. Weisfeiler and leman go machine learning: The story so far. *Journal of Machine Learning Research*, 24(1), 2023.

[MR24] Soroor Motie and Bijan Raahemi. Financial fraud detection using graph neural networks: A systematic review. *Expert Systems with Applications*, 240, 2024.

[MRF+19] Christopher Morris, Martin Ritzert, Matthias Fey, William L. Hamilton, Jan Eric Lenssen, Gaurav Rattan, and Martin Grohe. Weisfeiler and leman go neural: higher-order graph neural networks. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 4602–4609. AAAI Press, 2019.

[Mun57] James R. Munkres. Algorithms for the assignment and transportation problems. *Journal of the Society of Industrial and Applied Mathematics*, 5(1):32–38, 1957.

[NRB06] Michel Neuhaus, Kaspar Riesen, and Horst Bunke. Fast suboptimal algorithms for the computation of graph edit distance. In *Structural, Syntactic, and Statistical Pattern Recognition*, volume 4109 of *Lecture Notes in Computer Science*, pages 163–172. Springer, 2006.

[RB09] Kaspar Riesen and Horst Bunke. Approximate graph edit distance computation by means of bipartite graph matching. *Image Vision Comput.*, 27(7):950–959, 2009.

[RFFB15] Kaspar Riesen, Miquel Ferrer, Andreas Fischer, and Horst Bunke. Approximation of graph edit distance in quadratic time. In *Graph-Based Representations in Pattern Recognition*, pages 3–12. Springer, 2015.

[SHB19] Dagmar Stumpfe, Huabin Hu, and Jürgen Bajorath. Evolving concept of activity cliffs. *ACS Omega*, 4(11):14360–14368, 2019.

[SK98] Thomas Seidl and Hans-Peter Kriegel. Optimal multi-step k-nearest neighbor search. *SIGMOD Rec.*, 27(2):154–165, 1998.

[SSvL+11] Nino Shervashidze, Pascal Schweitzer, Erik Jan van Leeuwen, Kurt Mehlhorn, and Karsten M. Borgwardt. Weisfeiler-lehman graph kernels. *Journal of Machine Learning Research*, 12(77):2539–2561, 2011.

[SVP+09] Nino Shervashidze, SVN Vishwanathan, Tobias Petri, Kurt Mehlhorn, and Karsten Borgwardt. Efficient graphlet kernels for large graph comparison. In *Proceedings of the Twelth International Conference on Artificial Intelligence and Statistics*, volume 5, pages 488–495. PMLR, 2009.

[TGL+19] Matteo Togninalli, M. Elisabetta Ghisu, Felipe Llinares-López, Bastian Rieck, and Karsten M. Borgwardt. Wasserstein weisfeiler-lehman graph kernels. In *Proceedings of the 33rd International Conference on Neural Information Processing Systems*, pages 6436–6446. Curran Associates Inc., 2019.

[WDT+12] Xiaoli Wang, Xiaofeng Ding, Anthony K.H. Tung, Shanshan Ying, and Hai Jin. An efficient graph indexing method. In *28th International Conference on Data Engineering*, pages 210–221. IEEE, 2012.

[WL68] Boris Weisfeiler and Adrian Leman. A reduction of a graph to canonical form and an algebra arising during this reduction. *Nauchno-Technicheskaya Informatsia*, 2(9):12–16, 1968.

[WWYY12] Guoren Wang, Bin Wang, Xiaochun Yang, and Ge Yu. Efficiently indexing large sparse graphs for similarity search. *IEEE Transactions on Knowledge and Data Engineering*, 24(3):440–451, 2012.

[XHLJ19] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks? In *7th International Conference on Learning Representations*. OpenReview.net, 2019.

[YHC+18] Rex Ying, Ruining He, Kaifeng Chen, Pong Eksombatchai, William L. Hamilton, and Jure Leskovec. Graph convolutional neural networks for web-scale recommender systems. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, KDD '18, page 974–983. Association for Computing Machinery, 2018.

[YYGJ23] Hao Yuan, Haiyang Yu, Shurui Gui, and Shuiwang Ji. Explainability in graph neural networks: A taxonomic survey. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 45(5):5782–5799, 2023.

[ZTW+09] Zhiping Zeng, Anthony K. H. Tung, Jianyong Wang, Jianhua Feng, and Lizhu Zhou. Comparing stars: On approximating graph edit distance. *PVLDB*, 2:25–36, 2009.

[ZXL+13] Xiang Zhao, Chuan Xiao, Xuemin Lin, Qing Liu, and Wenjie Zhang. A partition-based approach to structure similarity search. *PVLDB*, 7:169–180, 2013.

[ZXLW12] Xiang Zhao, Chuan Xiao, Xuemin Lin, and Wei Wang. Efficient graph similarity joins with edit distance constraints. In *28th International Conference on Data Engineering*, pages 834–845. IEEE, 2012.

[ZZL+15] Weiguo Zheng, Lei Zou, Xiang Lian, Dong Wang, and Dongyan Zhao. Efficient graph similarity search over large graph databases. *IEEE Transactions on Knowledge and Data Engineering*, 27(4):964–978, 2015.

[ZZL+22] Wei Zhang, Fenghua Zhu, Yisheng Lv, Chang Tan, Wen Liu, Xin Zhang, and Fei-Yue Wang. Adapgl: An adaptive graph learning algorithm for traffic prediction based on spatiotemporal neural networks. *Transportation Research Part C: Emerging Technologies*, 139, 2022.

A Appended Papers

This appendix contains the publications, as well as additional information about them, such as the authors, publication venue, the DOI, the division of work among the authors, and the abstract.

A.1 Metric Indexing for Graph Similarity Search

Authors Franka Bause, David B. Blumenthal, Erich Schubert, and Nils M. Kriege

Publication Venue Published in *Similarity Search and Applications*, LNCS 13058 (2021).

DOI/URL https://doi.org/10.1007/978-3-030-89657-7_24

Authors Contributions

Franka Bause drafted, wrote, and revised the manuscript with input from the other authors, implemented the algorithms, and conducted the experimental evaluation.

David B. Blumenthal gave valuable insights regarding the Branch lower bound and helped to revise the manuscript.

Erich Schubert gave valuable feedback and helped incorporating the ELKI framework into the project.

Nils M. Kriege supervised the project, gave valuable feedback, and helped to revise the manuscript.

Reference in Thesis [BBSK21]

Abstract Finding the graphs that are most similar to a query graph in a large database is a common task with various applications. A widely-used similarity measure is the graph edit distance, which provides an intuitive notion of similarity and naturally supports graphs with vertex and edge attributes. Since its computation is NP-hard, techniques for accelerating similarity search have been studied extensively. However, index-based approaches for this are almost exclusively designed for graphs with categorical vertex and edge labels and uniform edit costs. We propose a filter-verification framework for similarity search, which supports non-uniform edit costs for graphs with arbitrary attributes. We employ an expensive lower bound obtained by solving an optimal assignment problem. This filter distance satisfies the triangle inequality, making it suitable for acceleration by metric indexing. In subsequent stages, assignment-based upper bounds are used to avoid further exact distance computations. Our extensive experimental evaluation shows that a significant runtime advantage over both a linear scan and state-of-the-art methods is achieved.

Metric Indexing for Graph Similarity Search

Franka Bause[1(✉)], David B. Blumenthal[2], Erich Schubert[3], and Nils M. Kriege[1]

[1] Faculty of Computer Science, University of Vienna, Vienna, Austria
{franka.bause,nils.kriege}@univie.ac.at
[2] Department Artificial Intelligence in Biomedical Engineering (AIBE), Friedrich-Alexander University Erlangen-Nürnberg (FAU), Erlangen, Germany
david.b.blumenthal@fau.de
[3] Department of Computer Science, TU Dortmund University, Dortmund, Germany
erich.schubert@tu-dortmund.de

Abstract. Finding the graphs that are most similar to a query graph in a large database is a common task with various applications. A widely-used similarity measure is the graph edit distance, which provides an intuitive notion of similarity and naturally supports graphs with vertex and edge attributes. Since its computation is NP-hard, techniques for accelerating similarity search have been studied extensively. However, index-based approaches for this are almost exclusively designed for graphs with categorical vertex and edge labels and uniform edit costs. We propose a filter-verification framework for similarity search, which supports non-uniform edit costs for graphs with arbitrary attributes. We employ an expensive lower bound obtained by solving an optimal assignment problem. This filter distance satisfies the triangle inequality, making it suitable for acceleration by metric indexing. In subsequent stages, assignment-based upper bounds are used to avoid further exact distance computations. Our extensive experimental evaluation shows that a significant runtime advantage over both a linear scan and state-of-the-art methods is achieved.

Keywords: Graphs · Similarity search · Graph edit distance

1 Introduction

Graph-structured data is ubiquitous in many areas such as chemo- and bioinformatics or computer vision. A common task is to search a database containing a large number of graphs for those that are most similar to a given query graph. Such queries are submitted directly by the user or occur as subproblems in downstream

This work has been supported by the Vienna Science and Technology Fund (WWTF) through project VRG19-009, and Deutsche Forschungsgemeinschaft (DFG), project number 124020371, within the Collaborative Research Center SFB 876 "Providing Information by Resource-Constrained Analysis", SFB projects A2 and A6.

N. Reyes et al. (Eds.): SISAP 2021, LNCS 13058, pp. 323–336, 2021.
https://doi.org/10.1007/978-3-030-89657-7_24

machine learning algorithms. A widely accepted concept of graph similarity is the *graph edit distance*, which is the minimum cost for transforming one graph into the other by a sequence of edit operations. A strength of this measure is that it can elegantly be applied to graphs with vertex and edge attributes by defining the costs of edit operations adequately. For example, to compare protein graphs where vertices are annotated by the amino acid sequence of the secondary structure elements they represent, the Levenshtein distance was used [20].

However, the vast majority of efficient methods for similarity search in graph databases are limited to the special case where graphs have categorical labels and the costs of edit operations are uniform (either zero or one) [10,13,16,25,26,28–31]. A fairly recent development in this domain are neural graph embeddings, e.g. [19], which do not return exact similarity search results. For the pairwise computation of the graph edit distance, several exact approaches [10,15] and heuristics such as *bipartite graph matching* based on optimal vertex assignments [20] have been proposed, many of which support the graph edit distance in its full generality [15,20]. Several of these yield lower and upper bounds on the graph edit distance as a byproduct, which have just recently been compared systematically [4]. However, these lower bounds for the general graph edit distance are not yet widely used for similarity search in graph databases. For the methods based on optimal vertex assignments, it has only recently been shown how to derive a distance termed BRANCH that is guaranteed to be a lower bound and proven to satisfy the triangle inequality [2]. BRANCH has been shown to provide an excellent trade-off between tightness and running time [4].

We propose a filter-verification framework for similarity search, which supports the general graph edit distance with arbitrary metric edit costs and is hence suitable for graphs with any attributes comparable with a distance measure. We employ BRANCH as an initial filter accelerated by metric indexing. In the next stages, we derive upper bounds from the optimal assignment and improve them via local search to reduce the candidate set further, before performing verification by exact computation of the graph edit distance. We experimentally evaluate our approach on graphs with attributes and categorical labels showing the effectivity of the filter pipeline. The results show that our approach allows scalable similarity search in attributed graphs with non-uniform edit costs. For the special case of uniform edit costs, where competing methods are available, our approach is shown to outperform the state of the art.

2 Related Work

We discuss approaches for similarity search regarding the graph edit distance and methods for its pairwise exact or approximate computation.

2.1 Similarity Search in Graph Databases

Methods for similarity search in graph databases can be divided into two categories, depending on whether they compare overlapping or non-overlapping substructures. The methods *k-AT* [25], *CStar* [28], *Segos* [26] and *GSim* [30] belong

to the first category. These techniques are inspired by the *q-grams* used in the computation of the string edit distance. Either q-grams based on trees [25,26,28] or paths [30] are used. The methods *Pars* [29], *MLIndex* [16], and *Inves* [13] partition the graphs into non-overlapping substructures. They essentially obtain lower bounds based on the observation, that if x non-overlapping substructures of a database graph are not contained in the query graph, the graph edit distance is at least x. *Pars* uses a dynamic partitioning approach to achieve this, while *MLIndex* uses a multi-layered index to manage multiple partitions for each graph. *Inves* is a method used to verify whether the graph edit distance of two graphs is below a specified threshold by first trying to generate enough mismatching non-overlapping substructures. *Mixed* [31] combines the idea of q-grams and graph partitioning. These methods only allow uniform edit costs and are therefore not suitable for graphs with continuous attributes.

The concept of a *median graph* of a set of graphs regarding the graph edit distance has been studied extensively, see [3] and references therein. An application of median graphs is their use as routing objects in hierarchical index structures [3,23]. However, we are not aware of any concrete realization using this concept in a setting comparable to ours.

2.2 Pairwise Computation of the Graph Edit Distance

For computing the exact graph edit distance, both general-purpose algorithms [15] as well as approaches tailored to the verification step in graph databases have been proposed [9], which are usually based on depth- or breadth-first search [9,12], or integer linear programming [15]. As the exact computation of the graph edit distance is not feasible for larger graphs, many heuristics have been proposed, e.g., [2,4,11,14,18,20]. The properties of the dissimilarities obtained from these are in general not well investigated. For heuristics based on optimal vertex assignment [20], which are widely used in practice [24], a variant called Branch was recently studied thoroughly [2]. Branch is a lower bound on the graph edit distance, a pseudo-metric on graphs and supports arbitrary cost models (c.f., Sect. 4.1).

3 Preliminaries

We introduce the required basic concepts of graph theory and discuss database search with a focus on the metric space.

3.1 Graph Theory

A *graph* $G = (V, E, \mu, \nu)$ consists of a set of vertices $V(G)$, a set of edges $E(G) \subseteq V(G) \times V(G)$ between vertices of G, a labeling function for the vertices $\mu\colon V(G) \to L$, and a labeling function for the edges $\nu\colon E(G) \to L$. We discuss only undirected graphs and denote an edge between u and v by uv. The set of neighbors of a vertex $v \in V(G)$ is denoted by $N(v) = \{u \mid uv \in E(G)\}$.

Table 1. Notation for edit costs.

$c_v(u, v)$	Cost of substituting vertex u with vertex v (adjusting the label/attributes)
$c_v(u, \epsilon)$	Cost of deleting the isolated vertex u
$c_v(\epsilon, v)$	Cost of inserting the isolated vertex v
$c_e(uv, wx)$	Cost of substituting edge uv with edge wx (adjusting the label/attributes)
$c_e(uv, \epsilon)$	Cost of deleting the edge uv
$c_e(\epsilon, wx)$	Cost of inserting the edge wx

The set L can be categorical labels or arbitrary attributes including real-valued vectors and complex objects such as strings.

A measure commonly used to describe the dissimilarity of two graphs is the *graph edit distance*, which is the minimum cost for transforming one graph into the other using edit operations. An *edit operation* can be deleting or inserting an isolated vertex or an edge or relabeling any of the two. An *edit path* from graph G_1 to G_2 is a sequence of edit operations $(e_1, e_2, \dots)$ that transforms G_1 into G_2.

Definition 1 (Graph Edit Distance [20]). *Let c be an edit cost function assigning non-negative costs to edit operations. The* graph edit distance *between two graphs G_1 and G_2 is defined as*

$$d_{ged}(G_1, G_2) = \min\left\{ \sum_{i=1}^{k} c(e_i) \mid (e_1, \dots, e_k) \in \Upsilon(G_1, G_2) \right\},$$

where $\Upsilon(G_1, G_2)$ is the set of all possible edit paths from G_1 to G_2.

The costs of the different edit operations can be chosen as required for the specific use case, see Table 1 for our notation. If the edit costs are symmetric, non-negative, and strictly positive for each non-identical edit operation, the graph edit distance is a metric on graphs, treating graph isomorphism as identity [3]. Note that this holds even if the edit costs do not satisfy the triangle inequality (and hence are no metric), because the graph edit distance uses the edit path with minimal cost. In this work, we nonetheless assume that the edit costs respect the triangle inequality, i.e., we assume that the following inequalities hold[1]:

$$c_v(u, w) \leq c_v(u, v) + c_v(v, w) \qquad \forall (u, v, w) \in \mathcal{V}^3 \tag{1}$$

$$c_v(u, v) \leq c_v(u, \epsilon) + c_v(\epsilon, v) \qquad \forall (u, v) \in \mathcal{V}^2 \tag{2}$$

$$c_e(uv, yz) \leq c_e(uv, wx) + c_e(wx, yz) \qquad \forall (uv, wx, yz) \in \mathcal{E}^3 \tag{3}$$

$$c_e(uv, wx) \leq c_e(uv, \epsilon) + c_e(\epsilon, wx) \qquad \forall (uv, wx) \in \mathcal{E}^2 \tag{4}$$

Equations (1), (3), and (4) can be enforced via pre-processing without changing the graph edit distance and can hence be assumed to hold w.l.o.g. [5].

[1] For simplicity of notation, we have defined the costs on the vertices and edges instead of their labels. Hence, the sets $\mathcal{V}$ and $\mathcal{E}$ are all possible vertices and edges, respectively.

E.g., if we have $c(u,v) > c(u,w)+c(w,v)$, we can simply substitute $c(u,v)$ with $c(u,w)+c(w,v)$, because a minimum cost edit path cannot contain $c(u,v)$. The only remaining constraint, Equation (2), is met (to the best of our knowledge) in all applications where the graph edit distance is used to address real-world problems [24]. Computing the graph edit distance is NP-hard [29], rendering exact computation possible for small graphs only. There are several heuristics, many of which are based on solving an assignment problem.

Definition 2 (Assignment Problem). *Let A and B be two sets with $|A| = |B| = n$ and $c\colon A \times B \to \mathbb{R}$ a cost function. An* assignment *between A and B is a bijection $f\colon A \to B$. The cost of an assignment f is $c(f) = \sum_{a \in A} c(a, f(a))$. The* assignment problem *is to find an assignment with minimum cost.*

For an assignment instance (A, B, c), we denote the cost of an optimal assignment by $d^c_{\mathrm{oa}}(A, B)$. The assignment problem can be solved in cubic running time using a suitable implementation of the Hungarian method [8].

3.2 Searching in Databases

Databases provide means to store data to be able to retrieve, insert or change it efficiently. In the context of data analysis, retrieval (search) is usually the crucial operation on databases, because it will be performed much more often than updates. We focus on two important types of similarity queries when searching a database DB, the first of which is the *range query* for a radius r:

Definition 3 (Range Query). *A* range query $\mathrm{range}(q, r)$*, with query object q and range r, returns all objects in the database with a distance to the query object not exceeding the range, i.e.,* $\mathrm{range}(q, r) = \{o \in DB \mid d(o, q) \leq r\}$.

The second type of query considered here is the *k-nearest neighbor query*.

Definition 4 (k-Nearest Neighbor Query). *A k-nearest neighbor query (kNN query)* $\mathrm{NN}(q, k)$ *with query object q and parameter k returns the smallest set $\mathrm{NN}(q, k) \subseteq DB$, so that $|\mathrm{NN}(q, k)| \geq k$ and*

$$\forall o \in \mathrm{NN}(q,k), \forall o' \in DB \setminus \mathrm{NN}(q,k) : \quad d(o,q) < d(o',q).$$

In conjunction with range queries, it is preferable to return all the objects with a distance (including the query object, if part of the database), that does not exceed the distance to the kth neighbor, which may be more than k objects when tied. That yields an equivalency of the results of kNN queries and range queries, i.e., we have $\mathrm{range}(q,r) = \mathrm{NN}(q, |\mathrm{range}(q,r)|)$ and $\mathrm{NN}(q,k) = \mathrm{range}(q, r_k)$, where r_k is the maximum distance in $\mathrm{NN}(q,k)$. Provided that the distance used is a metric, both types of queries can be accelerated using metric indices. In our work, we use the *vantage point tree* (*vp-tree*) [27] as a classical method and the more recent *cover tree* [1], because they are available in the ELKI framework [21], but others could also be used. While the vp-tree is a height balanced binary

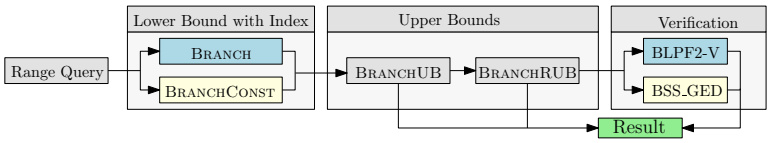


Fig. 1. Overview of the filter pipeline. For general metric edit costs the blue modules are used; yellow modules are more efficient for uniform (edge) edit costs.

tree dividing the data into *near* and *far* halves of the dataset based on the median distance from the vantage point, the cover tree controls the expansion rate by reducing the maximum radius in each level of the tree, branching out if necessary into multiple branches. In both trees, queries are performed top-down by traversing all paths that cannot be dismissed using the routing objects and employing the triangle inequality.

4 Efficient Filtering for the General Graph Edit Distance

We propose a filter pipeline for range queries regarding the graph edit distance following a common paradigm for expensive distances, see e.g. [28]. Here, lower bounds allow to filter out graphs that do not satisfy the query predicate. For the remaining candidates, upper bounds are evaluated to add them immediately to the result set without exact distance computation. Finally, in the verification step, the exact distance is computed for the remaining candidates only. Our approach starts with the optimal assignment based lower bound Branch accelerated by metric indexing. From the same optimal assignment, an upper bound is derived (BranchUB) and subsequently refined by local search (BranchRUB) before the remaining candidates are verified. The pipeline is illustrated in Fig. 1, the individual steps are described in the following.

4.1 Index-Accelerated Lower Bound Filtering

Several lower bounds on the graph edit distance have been proposed or can be derived from known heuristics, see [4]. One of the most effective lower bounds with an excellent trade-off between tightness and runtime is referred to as Branch.

Definition 5 (Branch Distance). *For two graphs G_1 and G_2 the* branch distance *is defined as $d_{branch}(G_1, G_2) = d^c_{\text{oa}}(V(G_1) \cup \varepsilon_1, V(G_2) \cup \varepsilon_2)$, where ε_i denotes a multiset of ϵ elements, so that $|V(G_i) \cup \varepsilon_i| = |V(G_1) \cup V(G_2)|$ for $i \in \{1, 2\}$, and*

$$c(u,v) = \begin{cases} 0 & \text{if } u = v = \epsilon \\ c_v(u,v) + d_e(u,v) & \text{if } u \neq \epsilon \text{ and } v \neq \epsilon \\ c_v(\epsilon,v) + 1/2 \cdot \sum_{n \in N(v)} c_e(\epsilon, vn) & \text{if } u = \epsilon \text{ and } v \neq \epsilon \\ c_v(u,\epsilon) + 1/2 \cdot \sum_{n \in N(u)} c_e(un, \epsilon) & \text{if } u \neq \epsilon \text{ and } v = \epsilon \end{cases},$$

with $d_e(u,v) = d^{c'}_{\text{oa}}(N(u) \cup \varepsilon_u, N(v) \cup \varepsilon_v)$, where $c'(w,x) = 1/2 \cdot c_e(uw, vx)$.

Remark 1. Note that, by using a customized version of the Hungarian algorithm, BRANCH can also be implemented in a slightly more efficient way, where only one dummy vertex ϵ is added to the vertex sets $V(G_1)$ and $V(G_2)$ (see [7] for details). In this paper, we use the classical implementation employed in [2,20], which corresponds to the characterization provided in Definition 5.

BRANCH has its origin in one of the most successful heuristics for the graph edit distance proposed by Riesen and Bunke [20]. However, in contrast to the original approach, it is guaranteed to underestimate the graph edit distance by dividing all edge costs by two to avoid that the cost of a single edge edit operation is counted twice, once for each endpoint [2]. Since an instance of the assignment problem on the vertices of the two graphs has to be solved, and for each vertex pair an assignment on their edges, BRANCH can be computed in $O(n^2\Delta^3 + n^3)$ time for graphs with n vertices and maximum degree Δ. In the case of uniform edge edit costs, d_e can be computed by multiset intersection of edge labels and the running time reduces to $O(n^3)$. This special case is referred to as BRANCHCONST [2]. It has been shown that, if the edit costs are metric, the branch distance is a pseudo-metric on graphs [2]. This allows to accelerate computing the candidate set w. r. t. this lower bound by employing metric indexing.

4.2 Upper Bound Filtering and Verification

From the solution of the assignment problem of BRANCH, an upper bound can be obtained by deriving the corresponding edit path [20], denoted BRANCHUB here. By definition of the graph edit distance, the cost of every edit path is an upper bound of the graph edit distance. Following [28], we refine the assignment by local search to gain a tighter upper bound. Starting with the assignment obtained for the lower bound, the mapping of two vertex pairs is iteratively swapped, and kept whenever it induces a cheaper edit path, until there is no improvement. We refer to the refined upper bound obtained from the BRANCH assignment as BRANCHRUB.

Eventually, the graphs that were neither filtered out by the lower bound nor approved by the upper bounds are verified by exact graph edit distance computation. We use *BSS_GED* [10] for datasets with discrete labels and uniform costs and *BLPF2-V* otherwise. The latter is based on the integer programming formulation F2 of [15] with the additional constraint that the objective function does not exceed the threshold to allow for early termination.

4.3 Nearest-Neighbor Queries

For kNN queries it is not possible to separate the different steps of the filter pipeline as clearly as shown in Fig. 1. We realize kNN queries using the *optimal multi-step k-nearest neighbor search* algorithm [22]. The database graphs are scanned in ascending order according the lower bound BRANCH regarding the query graph. For each graph, the exact graph edit distance is computed and the k

Table 2. Datasets and their statistics [17]. Some datasets contain graphs with labeled or attributed vertices and edges, as can be seen in the last two columns.

Name	|Graphs|	avg |Vertices|	avg |Edges|	Labels (V/E)	Attributes (V/E)
Cuneiform	267	21.27	44.80	+/+	+/+
Fingerprint	2800	5.42	4.42	−/−	+/+
Letter-high	2250	4.67	4.50	−/−	+/−
Letter-low	2250	4.68	3.13	−/−	+/−
MUTAG	188	17.93	19.79	+/+	−/−
PTC_FM	349	14.11	14.48	+/+	−/−
QM9	129433	18.03	18.63	−/−	+/+

graphs with the smallest exact graph edit distance are maintained. Once we have found at least k objects with an exact distance smaller than the lower bound of all remaining objects, the search can be terminated. This is optimal in the sense that none of the exact distance computations could have been avoided [22]. Accessing the graphs ordered regarding the BRANCH lower bound can be achieved naïvely by sorting all graphs, or by using suitable metric index structures.

5 Experimental Evaluation

In this section, we experimentally address the following research questions:

- Q1 What speed-up in range queries can be achieved when using metric indices compared to a linear scan of the database?
- Q2 How effective are the individual lower and upper bounds in our pipeline?
- Q3 Can the proposed filter pipeline compete with state-of-the-art methods for uniform edit costs?
- Q4 What speed-up in kNN queries can be achieved when using metric indices?
- Q5 Does the proposed filter pipeline scale to a very large dataset?

5.1 Setup

As metric index we chose the *vp-tree* as a classical method and the *cover tree* as a state-of-the-art approach. For both we used the implementation provided by ELKI [21] with a sample size of 5 for the vp-tree and an expansion rate of 1.2 for the cover tree.

For a comparison in databases containing graphs with categorical labels, we used *MLIndex* [16] and *GSim* [30], since the former is considered state-of-the-art, while the latter provided much better results in our experiments. For *MLIndex* the number of partitions was set to threshold + 1 and in *GSim* all provided filters were used. We used the implementations provided by the authors. In addition, we used *CStar* [28], which follows a filter-verification approach related to ours. For verification we used *BSS_GED* [10] and *BLPF2-V* [15] with the Gurobi solver.

We conducted experiments on a wide range of real-world datasets with different characteristics, see Table 2. The costs of inserting, deleting or relabeling

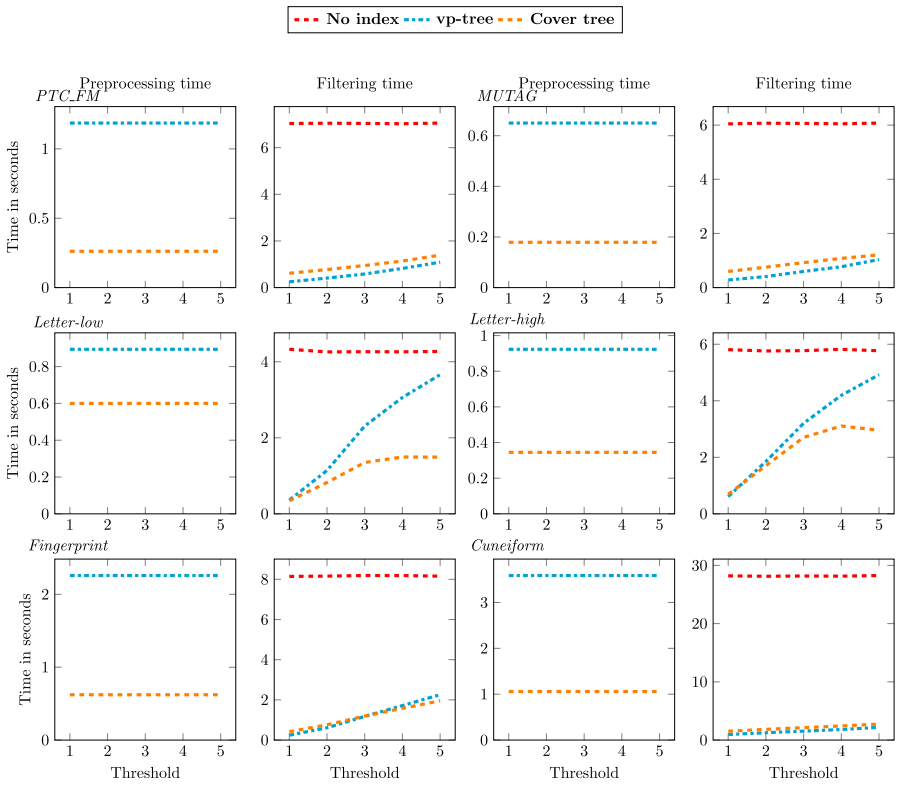


Fig. 2. Runtime comparison for filtering 100 range queries using BRANCH with thresholds 1 to 5 and preprocessing time for constructing the index.

a vertex or edge with a categorical label were set to 1, which is equivalent to the fixed setting in *MLIndex*, *GSim*, and *CStar*. For continuous attributes, the Euclidean distance was used to define the relabeling cost. For simplicity, we did not use domain-specific distances. Continuous attributes were normalized to the range $[0, 1]$ (separately for each dimension), to make distances roughly comparable between different datasets.

5.2 Results

We report on our findings regarding the above research questions.

Q1: Speed-up of range queries through metric indices. We first investigate how much of a speed-up can be achieved by using an index structure when filtering candidates for a range query by a lower bound. We randomly sampled 100 graphs from the dataset to be queries and then performed lower bound filtering without an index, using the cover tree, and the vp-tree.

Figure 2 shows the time needed for filtering 100 range queries (each with thresholds 1 to 5) and additionally the preprocessing time for index construction. The runtime does not depend on the given threshold for a linear scan, but

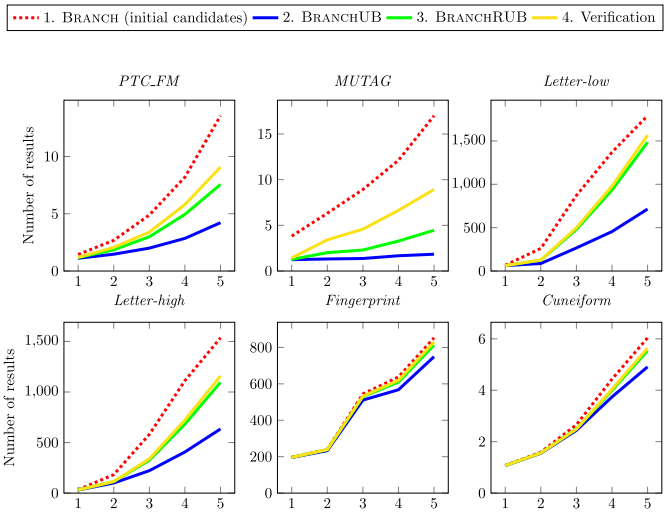


Fig. 3. Average number of initial candidates (dashed) and hits identified in the different stages of the filter pipeline for each threshold.

increases for the metric indices with the threshold, in particular for the *Letter*-datasets. It can be seen that, while on most datasets both index methods provide the same runtime benefit for filtering, the cover tree is much faster in preprocessing than the vp-tree. The runtime advantage on the *Letter*-datasets is quite small for larger thresholds. The runtime of the index structures directly corresponds to the number of Branch distance computations. Compared to the cover tree, the vp-tree requires many more distance computations in the preprocessing due to the chosen sample size. In general, the runtime corresponds to the number of candidates, which we investigate in the following.

Q2: Filter pipeline. In this experiment we investigate how the candidate and result set are updated during filtering. Figure 3 shows the average number of candidates for Branch and the number of results after each step for 100 range queries. When comparing the size of the candidate sets with the results of the previous experiment, it can be seen, that the runtime for filtering highly depends on the number of candidates. For some datasets almost all candidates remaining after the upper bound filtering are not results. This indicates that improvement is possible with tighter lower bounds. In general, BranchRUB manages to report almost all results, except in dataset *MUTAG*.

Q3: Comparison with state-of-the-art methods. Many methods for similarity search in graph databases limited to uniform edit costs have been proposed. We compare to *MLIndex* [16], *CStar* [28] and *GSim* [30]. We used *BSS_GED* [10] for a fast verification in our filter pipeline, as well as in *CStar*. The implementations of *MLIndex* and *GSim* contain their own verification algorithm. Figure 4 shows the runtime for preprocessing and filtering as well as the total query time

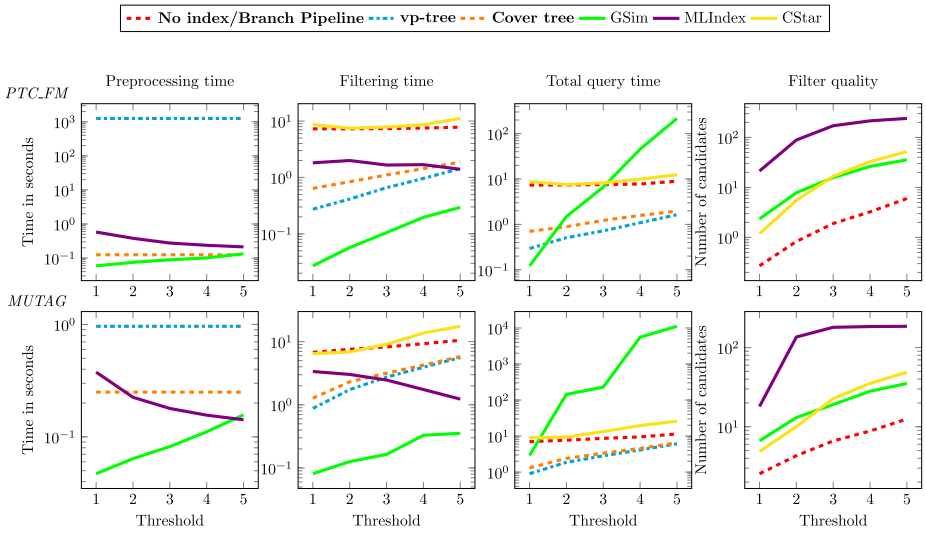


Fig. 4. Runtime for answering 100 range queries and average number of candidates remaining after applying all filters for thresholds 1 to 5. Our approaches are shown with dashed lines and are marked bold in the legend. For *MLIndex* no verification time is given, since it did not finish within the time limit of 2 days.

including filtering and verification for 100 range queries. The average number of candidates remaining after application of all filters in the different methods is also shown. Only these need to be verified by exact graph edit distance computation. For BRANCH only one line is shown, since linear scan, *vp-tree* and the *cover tree* variant apply the same filters and generate the same candidates.

MLIndex produces the largest candidate set, and did not finish the verification process in the time limit. It can be seen that, while *GSim* is quite fast in preprocessing and filtering, the verification step takes a long time. This is due to a combination of a slower verification algorithm and a higher number of candidates that have to be verified. The results indicate that, even when using *BSS_GED* for verification, the approach would not be competitive with the *cover tree* due to the high number of candidates. *CStar* needs much more time for filtering and cannot filter out as many candidates leading also to a higher verification time. Interestingly, the time for verification does not increase proportionally to the number of candidates, which might indicate, that the verification algorithm needs more time to verify certain *difficult* graphs.

Q4: Speed-up of kNN queries through metric indices. We investigate how much of a speed-up can be achieved by using an index structure compared to not using one, when answering kNN queries using the optimal multi-step k-nearest neighbor search, cf. Sect. 4.3. We randomly sampled 20 graphs from the dataset to be queries and then used the cover tree as well as the vp-tree as the underlying metric index to compare them. Figure 5 shows the time needed for answering 20 kNN queries, each with $k \in \{1, \ldots, 5\}$ (excluding preprocessing). Since in the

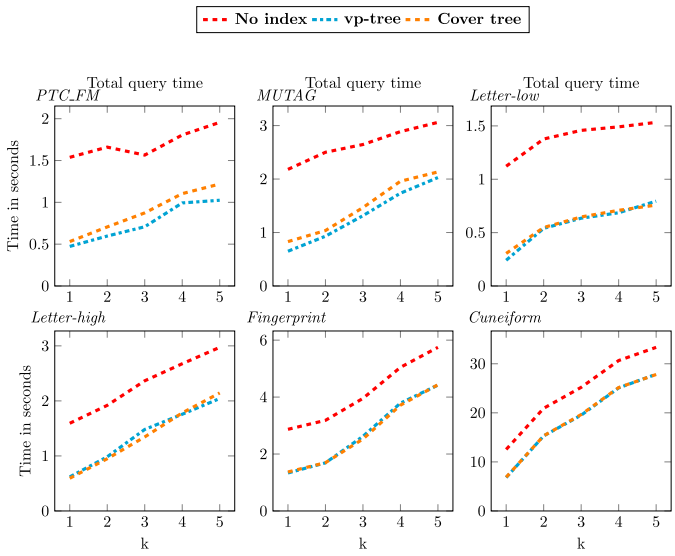


Fig. 5. Runtime comparison for answering 20 kNN queries using Branch and $k \in \{1, \ldots, 5\}$.

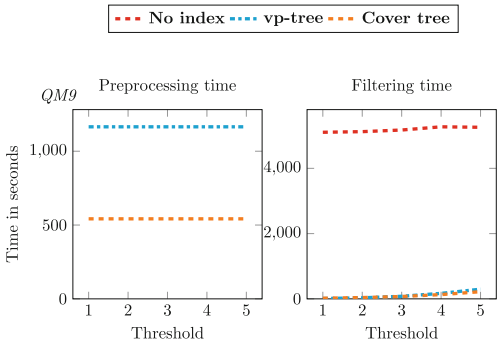


Fig. 6. Runtime comparison for preprocessing and filtering 100 range queries in the dataset *QM9* using Branch and thresholds 1 to 5.

optimal multi-step k-nearest neighbor search, the candidates have to be verified during search, before further candidates are explored, the runtime also includes the time needed for verification. It can be seen that, again both index structures provide the same runtime benefit. Taking into account the preprocessing time however, the cover tree has a clear advantage over the vp-tree.

Q5: Similarity search in a large dataset. We investigate the scalability of our approach on the dataset *QM9* with 129 433 graphs with attributed vertices and edges. The results shown in Fig. 6 confirm the high preprocessing time of the vp-tree compared to the cover tree. Both index methods achieve a significant advantage over a linear scan in filtering by reducing the running time by several orders of magnitude depending on the selectivity of the query.

6 Conclusions

We have shown that the recently studied lower and upper bounds on the graph edit distance can be employed to realize scalable graph similarity search in a filter-verification framework accelerated by metric indexing. Our approach supports attributed graphs without restrictions of edit costs. For the extensively studied special case of graphs with discrete labels and uniform edit costs, our approach was shown experimentally to outperform the state-of-the-art methods.

There are several directions of future work to improve the filter-verification pipeline further. Our tightest upper bound was obtained via local search using a straightforward approach. More sophisticated techniques have been proposed recently [6] and can be incorporated to reduce verification. For the verification step, tailored methods that benefit from the already obtained assignment or the upper and lower bound can be developed. A well-known phenomenon of metric trees is that their effectivity decreases with increasing intrinsic dimensionality of the data/distance. Therefore, a suitable lower bound should not only be efficiently computed and tight, but ideally also have a low intrinsic dimensionality. Studying this property for the available lower bounds remains future work. Finally, recent advances in median graph computation [3] suggest to compute routing objects instead of using database graphs. An experimental comparison to such orthogonal approaches remains future work.

References

1. Beygelzimer, A., Kakade, S.M., Langford, J.: Cover trees for nearest neighbor. In: International Conference Machine Learning, ICML, vol. 148, pp. 97–104 (2006)
2. Blumenthal, D.B., Gamper, J.: Improved lower bounds for graph edit distance. IEEE Trans. Knowl. Data Eng. **30**(3), 503–516 (2018)
3. Blumenthal, D.B., Boria, N., Bougleux, S., Brun, L., Gamper, J., Gaüzère, B.: Scalable generalized median graph estimation and its manifold use in bioinformatics, clustering, classification, and indexing. Inf. Syst. **100**, 101766 (2021)
4. Blumenthal, D.B., Boria, N., Gamper, J., Bougleux, S., Brun, L.: Comparing heuristics for graph edit distance computation. VLDB J. **29**(1), 419–458 (2019). https://doi.org/10.1007/s00778-019-00544-1
5. Blumenthal, D.B., Gamper, J.: On the exact computation of the graph edit distance. Pattern Recogn. Lett. **134**, 46–57 (2020)
6. Boria, N., Blumenthal, D.B., Bougleux, S., Brun, L.: Improved local search for graph edit distance. Pattern Recogn. Lett. **129**, 19–25 (2020)
7. Bougleux, S., Gaüzère, B., Blumenthal, D.B., Brun, L.: Fast linear sum assignment with error-correction and no cost constraints. Pattern Recogn. Lett. **134**, 37–45 (2020)
8. Burkard, R.E., Dell'Amico, M., Martello, S.: Assignment Problems. SIAM, Philadelphia (2012)
9. Chang, L., Feng, X., Lin, X., Qin, L., Zhang, W., Ouyang, D.: Speeding up GED verification for graph similarity search. In: International Conference Data Engineering, ICDE, pp. 793–804 (2020)
10. Chen, X., Huo, H., Huan, J., Vitter, J.S.: An efficient algorithm for graph edit distance computation. Knowl. Based Syst. **163**, 762–775 (2019)

11. Gouda, K., Arafa, M., Calders, T.: BFST_ED: a novel upper bound computation framework for the graph edit distance. In: Amsaleg, L., Houle, M.E., Schubert, E. (eds.) SISAP 2016. LNCS, vol. 9939, pp. 3–19. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-46759-7_1
12. Gouda, K., Hassaan, M.: CSI_GED: an efficient approach for graph edit similarity computation. In: International Conference Data Engineering, ICDE, pp. 265–276 (2016)
13. Kim, J., Choi, D., Li, C.: Inves: incremental partitioning-based verification for graph similarity search. In: Extending Database Technology, EDBT, pp. 229–240 (2019)
14. Kriege, N.M., Giscard, P., Bause, F., Wilson, R.C.: Computing optimal assignments in linear time for approximate graph matching. In: ICDM, pp. 349–358 (2019)
15. Lerouge, J., Abu-Aisheh, Z., Raveaux, R., Héroux, P., Adam, S.: New binary linear programming formulation to compute the graph edit distance. Pattern Recogn. **72**, 254–265 (2017)
16. Liang, Y., Zhao, P.: Similarity search in graph databases: A multi-layered indexing approach. In: International Conference Data Engineering, ICDE, pp. 783–794 (2017)
17. Morris, C., Kriege, N.M., Bause, F., Kersting, K., Mutzel, P., Neumann, M.: TUDataset: a collection of benchmark datasets for learning with graphs. In: ICML 2020 Workshop on Graph Representation Learning and Beyond, GRL+ (2020)
18. Neuhaus, M., Riesen, K., Bunke, H.: Fast suboptimal algorithms for the computation of graph edit distance. In: Structural, Syntactic, and Statistical Pattern Recognition. pp. 163–172, August 2006
19. Qin, Z., Bai, Y., Sun, Y.: Ghashing: semantic graph hashing for approximate similarity search in graph databases. In: ACM SIGKDD, pp. 2062–2072 (2020)
20. Riesen, K., Bunke, H.: Approximate graph edit distance computation by means of bipartite graph matching. Image Vis. Comput. **27**(7), 950–959 (2009)
21. Schubert, E., Zimek, A.: ELKI: a large open-source library for data analysis - ELKI release 0.7.5 "Heidelberg". CoRR abs/1902.03616 (2019)
22. Seidl, T., Kriegel, H.: Optimal multi-step k-nearest neighbor search. In: SIGMOD International Conference Management of Data, pp. 154–165 (1998)
23. Serratosa, F., Cortés, X., Solé-Ribalta, A.: Graph database retrieval based on metric-trees. In: SSPR, pp. 437–447 (2012)
24. Stauffer, M., Tschachtli, T., Fischer, A., Riesen, K.: A survey on applications of bipartite graph edit distance. In: GbRPR, pp. 242–252 (2017)
25. Wang, G., Wang, B., Yang, X., Yu, G.: Efficiently indexing large sparse graphs for similarity search. IEEE Trans. Knowl. Data Eng. **24**(3), 440–451 (2012)
26. Wang, X., Ding, X., Tung, A., Ying, S., Jin, H.: An efficient graph indexing method. In: International Conference Data Engineering, ICDE (2012)
27. Yianilos, P.N.: Data structures and algorithms for nearest neighbor search in general metric spaces. In: SODA, pp. 311–321 (1993)
28. Zeng, Z., Tung, A.K.H., Wang, J., Feng, J., Zhou, L.: Comparing stars: on approximating graph edit distance. Proc. VLDB Endow. **2**(1), 25–36 (2009)
29. Zhao, X., Xiao, C., Lin, X., Liu, Q., Zhang, W.: A partition-based approach to structure similarity search. Proc. VLDB Endow. **7**(3), 169–180 (2013)
30. Zhao, X., Xiao, C., Lin, X., Wang, W.: Efficient graph similarity joins with edit distance constraints. In: International Conference Data Engineering, ICDE (2012)
31. Zheng, W., Zou, L., Lian, X., Wang, D., Zhao, D.: Efficient graph similarity search over large graph databases. IEEE Trans. Knowl. Data Eng. **27**(4), 964–978 (2015)

A.2 EmbAssi: Embedding Assignment Costs for Similarity Search in Large Graph Databases

Authors Franka Bause, Erich Schubert, and Nils M. Kriege

Publication Venue Published in *Data Mining and Knowledge Discovery* 36 (2022).

DOI/URL https://doi.org/10.1007/s10618-022-00850-3

Authors Contributions

Franka Bause developed the lower bounds and theoretical results, drafted, wrote, and revised the manuscript with input from the other authors, implemented the algorithms, and conducted the experimental evaluation.

Erich Schubert gave valuable feedback and helped incorporating the ELKI framework into the project.

Nils M. Kriege supervised the project, gave valuable feedback, and helped to revise the manuscript.

Distinctions from Master's Thesis The following contributions were made additionally to the content of the master's thesis [Bau20]:

- The discussion of related work was significantly expanded.
- The proofs for the lower bounds (Propositions 1–5) were revised and formalized.
- A theoretical analysis was conducted on the relationships between the various bounds (Proposition 6).
- Theoretical time complexity was examined in detail (Propositions 7 and 8).
- Methods for further improving the efficiency of the lower bounds were explored (Section 5.1).
- The framework was extended to enable the combination of different bounds, and the code was extensively refactored to improve both efficiency and readability.
- Additional experiments were conducted with a broader range of competitor methods and datasets.

Reference in Thesis [BSK22]

Abstract

The graph edit distance is an intuitive measure to quantify the dissimilarity of graphs, but its computation is NP-hard and challenging in practice. We introduce methods for answering nearest neighbor and range queries regarding this distance efficiently for large databases with up to millions of graphs. We build on the filter-verification paradigm, where lower and upper bounds are used to reduce the number of exact computations of the graph edit distance. Highly effective bounds for this involve solving a linear assignment problem for each graph in the database, which is prohibitive in massive datasets. Index-based approaches typically provide only weak bounds leading to high computational costs verification. In this work, we derive novel lower bounds for efficient filtering from restricted assignment problems, where the cost function is a tree metric. This special case allows embedding the costs of optimal assignments isometrically into ℓ_1 space, rendering efficient indexing possible. We propose several lower bounds of the graph edit distance obtained from tree metrics reflecting the edit costs, which are combined for effective filtering. Our method termed EmbAssi can be integrated into existing filter-verification pipelines as a fast and effective pre-filtering step. Empirically we show that for many real-world graphs our lower bounds are already close to the exact graph edit distance, while our index construction and search scales to very large databases.

Data Mining and Knowledge Discovery (2022) 36:1728–1755
https://doi.org/10.1007/s10618-022-00850-3

EmbAssi: embedding assignment costs for similarity search in large graph databases

Franka Bause[1,2] · Erich Schubert[3] · Nils M. Kriege[1,4]

Received: 8 October 2021 / Accepted: 21 June 2022 / Published online: 16 July 2022

Abstract

The graph edit distance is an intuitive measure to quantify the dissimilarity of graphs, but its computation is NP-hard and challenging in practice. We introduce methods for answering nearest neighbor and range queries regarding this distance efficiently for large databases with up to millions of graphs. We build on the filter-verification paradigm, where lower and upper bounds are used to reduce the number of exact computations of the graph edit distance. Highly effective bounds for this involve solving a linear assignment problem for each graph in the database, which is prohibitive in massive datasets. Index-based approaches typically provide only weak bounds leading to high computational costs verification. In this work, we derive novel lower bounds for efficient filtering from restricted assignment problems, where the cost function is a tree metric. This special case allows embedding the costs of optimal assignments isometrically into ℓ_1 space, rendering efficient indexing possible. We propose several lower bounds of the graph edit distance obtained from tree metrics reflecting the edit costs, which are combined for effective filtering. Our method termed *EmbAssi* can be integrated into existing filter-verification pipelines as a fast and effective pre-filtering step. Empirically we show that for many real-world graphs our lower bounds are already close to the exact graph edit distance, while our index construction and search scales to very large databases.

Responsible editor: Albrecht Zimmermann and Peggy Cellier

✉ Franka Bause
franka.bause@univie.ac.at

✉ Nils M. Kriege
nils.kriege@univie.ac.at

Erich Schubert
erich.schubert@tu-dortmund.de

1 Faculty of Computer Science, University of Vienna, Vienna, Austria

2 UniVie Doctoral School Computer Science, University of Vienna, Vienna, Austria

3 Department of Computer Science, TU Dortmund University, Dortmund, Germany

4 Research Network Data Science, University of Vienna, Vienna, Austria

Keywords Graph edit distance · Similarity search · Index

1 Introduction

In various applications such as cheminformatics (Garcia-Hernandez et al. 2019), bioinformatics (Stöcker et al. 2019), computer vision (Xiao et al. 2013) and social network analysis, complex structured data arises, which can be naturally represented as graphs. To analyze large amounts of such data, meaningful measures of (dis)similarity are required. A widely accepted approach is the *graph edit distance*, which measures the dissimilarity of two graphs in terms of the total cost of transforming one graph into the other by a sequence of edit operations. This concept is appealing because of its intuitive and comprehensible definition, its flexibility to adapt to different types of graphs and annotations, and the interpretability of the dissimilarity measure. However, computing the graph edit distance for a pair of graphs is NP-hard (Zeng et al. 2009) and challenging in practice even for small graphs. This renders similarity search regarding the graph edit distance in databases difficult, which is relevant in many applications. A prime example is a molecular information system, which often contains millions of graphs representing small molecules. A standard task in computational drug discovery is similarity search in such databases, for which the concept of graph edit distance has proven useful (Garcia-Hernandez et al. 2019). However, the extensive use of graph-based methods in such systems is still hindered by the computational burden, especially in comparison to embedding-based techniques, for which efficient similarity search is well studied (Nasr et al. 2010). Moreover, similarity search is the fundamental problem when using the graph edit distance in downstream supervised or unsupervised machine learning methods such as k-nearest neighbors classification. Promising results have been reported for classifying graphs from diverse applications representing, e.g., small molecules (Kriege et al. 2019), petroglyphs (Seidl et al. 2015), or cuneiform signs (Kriege et al. 2018). However, this approach does not readily scale to large datasets, where embedding-based methods such as graph kernels (Kriege et al. 2020) and graph neural networks (Wu et al. 2021) have become the dominating techniques.

Algorithms for exact (Gouda and Hassaan 2016), (Lerouge et al. 2017), (Chang et al. 2020), (Chen et al. 2019) or approximate (Neuhaus et al. 2006), (Riesen and Bunke 2009), (Kriege et al. 2019) graph edit distance computation have been extensively studied. They are typically optimized for pairwise comparison but can be accelerated in cases when a distance cutoff is given as part of the input. While not directly suitable for searching large databases, these algorithms are used in the verification step after a set of candidates has been obtained by filtering. In the filtering step, lower bounds on the graph edit distance are typically used to eliminate graphs that cannot satisfy the distance threshold, while upper bounds are used to add graphs to the answer set without the need for verification. Several techniques following this paradigm have been proposed, see Table 1 for an overview. An important characteristic for scalability is whether these techniques use an index to avoid scanning every graph in the database. This is not directly possible for many of the existing bounds on the graph edit distance, which were often studied in other contexts. A recent systematic comparison of existing

Table 1 Overview of methods for similarity search in graph databases

Name	Ref.	Description	Bounds	Index	Exact
CStar	(Zeng et al. 2009)	Optimal assignments on star structures	lower/upper	no	yes
GSim	(Zhao et al. 2012)	Approximation using path-based q-grams	lower	yes	yes
Segos	(Wang et al. 2012)	Two-level inverted index	lower/upper	yes	yes
k-AT	(Wang et al. 2012)	Index over tree-based q-grams	lower	yes	yes
Pars	(Zhao et al. 2013)	Dynamic partitioning of graphs	lower	yes	yes
Mixed	(Zheng et al. 2015)	Partitioning and q-gram based filtering	lower	yes	yes
MLIndex	(Liang and Zhao 2017)	Partitioning graphs in a multi-layered Index	lower	yes	yes
Inves	(Kim et al. 2019)	Incremental partitioning used in verification	lower/upper	no	yes
BSS_GED	(Chen et al. 2019)	Filtering and efficient verification algorithm	lower	no	yes
GHashing	(Qin et al. 2020)	Approximate filter via hashing and GNNs	—	yes	no
EmbAssi		Embedding assignment costs	lower	yes	yes

bounds (Blumenthal et al. 2019) shows that there is a trade-off between the efficiency of computation and the tightness of lower and upper bounds. Lower bounds based on linear programming relaxations and solutions of the linear assignment problem were found to be most effective. However, the computation of such bounds requires solving non-trivial optimization problems and is inefficient compared to computing standard distances on vectors. Moreover, their combination with well-studied indices for vector or metric data is often not feasible because they do not satisfy the necessary properties such as being metric or embeddable into vector space. (Qin et al. 2020) concluded, that methods without an index do not scale well to very large databases, while those with an index often provide only loose bounds leading to a high computational cost for verification. To overcome this, they proposed an inexact filtering mechanism based on hashing, which cannot guarantee a complete answer set. We show that exact similarity search in very large databases using the filter-verification paradigm is possible. We achieve this by developing tight lower bounds based on assignment costs which are embedded into a vector space for index-based acceleration.

Our Contribution We develop multiple efficiently computable tight lower bounds for the graph edit distance, that allow exact filtering and can be used with an index for scalability to large databases. Our techniques are shown to achieve a favorable trade-

off between efficiency and effectivity in filtering. Specifically, we make the following contributions.

(1) *Embedding Assignment Costs.* We build on a restricted version of the combinatorial assignment problem between two sets, where the ground costs for assigning individual elements are a tree metric. With this constraint, the cost of an optimal assignment equals the ℓ_1 distance between vectors derived from the sets and the weighted tree representing the metric (Kriege et al. 2019). We show that these vectors can be computed in linear time and optimized for combination with well-studied indices for vector data (Schubert and Zimek 2019).

(2) *Lower Bounds.* We formulate several assignment-based distance functions for graphs that are proven to be lower bounds on the graph edit distance. We show that their ground cost functions are tree metrics and derive the corresponding trees, from which suitable vector representations are computed. We propose bounds supporting uniform as well as non-uniform edit cost models for vertex labels. Further bounds based on vertex degrees and labeled edges are introduced, some of which can be combined to obtain tighter lower bounds. We analyze the proposed lower bounds and formally relate them to existing bounds from the literature.

(3) *EmbAssi.* We use the vector representation for similarity search in graph databases following the filter-verification paradigm, building upon established indices for the Manhattan (ℓ_1) distance on vectors. Our approach supports range queries as well as k-nearest neighbor search using the optimal multi-step k-nearest neighbor search algorithm (Seidl and Kriegel 1998). This allows employing our approach in downstream machine learning and data mining methods such as nearest neighbors classification, local outlier detection (Schubert et al. 2014), or density-based clustering (Ester et al. 1996).

(4) *Experimental Evaluation.* We show that, while the proposed bounds are often close to or even outperform state-of-the-art bounds (Blumenthal et al. 2019), (Zeng et al. 2009), they can be computed much more efficiently. In the filter-verification framework, our approach obtains manageable candidate sets for verification in a very short time even in databases with millions of graphs, for which most competitors fail. Our approach supports efficient construction of an index used for all query thresholds and is, compared to several competitors (Liang and Zhao 2017), (Zhao et al. 2012), not restricted to connected graphs with a certain minimum size. We show that our approach can be combined with more expensive lower and upper bounds in a subsequent step to further reduce overall query time.

2 Related work

We summarize the related work on similarity search in graph databases and graph edit distance computation and conclude by a discussion motivating our approach.

2.1 Similarity search in graph databases

Several methods for accelerating similarity search in graph databases have been proposed, see Table 1. Most approaches follow the filter-verification paradigm and rely on lower and upper bounds of the graph edit distance. These techniques focus almost exclusively on range queries and assume a uniform cost function for graph edit operations. Most of the methods suitable for similarity search can be divided into two categories depending on whether they compare overlapping or non-overlapping substructures.

Representatives of the first category are *k-AT* (Wang et al. 2012), *CStar* (Zeng et al. 2009), *Segos* (Wang et al. 2012) and *GSim* (Zhao et al. 2012). These methods are inspired by the concept of *q-grams* commonly used for string matching. In (Wang et al. 2012) tree-based q-grams on graphs were proposed. The *k-adjacent tree* of a vertex $v \in V(G)$, denoted k-AT(v), is defined as the top-k level subtree of a breadth-first search tree in G, starting with vertex v. For example, the 1-AT(v) is a tree rooted at v with the neighbors of v as children. These trees can be generated for each vertex of a graph and the graph can then be represented as the set of its k-ATs. Lower bounds for filtering are computed from these representations, which are organized in an inverted index. *CStar* (Zeng et al. 2009) is a method for computing an upper and lower bound on the graph edit distance using so-called *star representations* of graphs, which consist of a 1-AT for each vertex, called *star*. An optimal assignment between the star representations of graphs regarding a ground cost function on stars the assignment cost yields a lower bound (Zeng et al. 2009). An upper bound can be obtained by using the cost of an edit path induced by the optimal assignment. *Segos* (Wang et al. 2012) also uses these stars as (overlapping) substructures, but enhances the computation of the mapping distance and makes use of a two-layered index for range queries. Another view on q-grams is given by the *GSim* method (Zhao et al. 2012), which uses path-based q-grams, i.e., simple path of length q, instead of stars. Since the number of path-based q-grams affected by an edit operation is lower than the number of tree-based q-grams, the derived lower bound is tighter (Zhao et al. 2012).

The second category includes *Pars* (Zhao et al. 2013), *MLIndex* (Liang and Zhao 2017) and *Inves* (Kim et al. 2019), which partition the graphs into non-overlapping substructures. They essentially obtain lower bounds based on the observation, that if x partitions of a database graph are not contained in the query graph, the graph edit distance is at least x. *Pars* uses a dynamic partitioning approach to exploit this, while *MLIndex* uses a multi-layered index to manage multiple partitionings for each graph. *Inves* is a method used to verify whether the graph edit distance of two graphs is below a specified threshold by first trying to generate enough mismatching partitions. *Mixed* (Zheng et al. 2015) combines the idea of q-grams and graph partitioning. First, a lower bound that uses the same idea as *Inves* (Kim et al. 2019), but a different approach to find mismatching partitions, is proposed. Another lower bound based on so-called branch structures (a vertex and its adjacent edges without the opposite vertex) is combined with the first one to gain an even tighter lower bound. This bound can be generalized to non-uniform edit costs and is referred to as *Branch* (Blumenthal et al. 2019). Recently,

it has been proven that this bound is metric and its combination with an index to speed up similarity search for attributed graphs has been proposed (Bause et al. 2021).

2.1.1 Pairwise computation of the graph edit distance

In the verification step, the remaining candidates have to be validated by computing the exact graph edit distance. Both general-purpose algorithms (Lerouge et al. 2017) as well as approaches tailored to the verification step have been proposed (Chang et al. 2020), which are usually based on depth- or breadth-first search (Gouda and Hassaan 2016), (Chang et al. 2020) or integer linear programming (Lerouge et al. 2017).

On large graphs, these methods are not feasible and approximations are used (Neuhaus et al. 2006), (Chen et al. 2019), (Riesen and Bunke 2009), (Kriege et al. 2019). These can be obtained from the exact approaches, e.g., using beam search (Neuhaus et al. 2006) or linear programming relaxations (Blumenthal et al. 2019). *BeamD* (Neuhaus et al. 2006) finds a sub-optimal edit path following the A^* algorithm by extending only a fixed number of partial solutions. A state-of-the-art approach is *BSS_GED* (Chen et al. 2019), which reduces the search space based on beam stack search. It is not only used for computation of the exact graph edit distance, but also for similarity search by filtering with lower bounds during a linear database scan. Recently, an approach using neural networks to improve the performance of the beam search algorithm was proposed (Yang and Zou 2021).

A successful technique referred to as *bipartite graph matching* (Riesen and Bunke 2009) obtains a sub-optimal edit path from the solution of an optimal assignment between the vertices where the ground costs also encode the local edge structure. The assignment problem is solved in cubic time using Hungarian-type algorithms (Burkard et al. 2012), (Munkres 1957) or in quadratic time using simple greedy strategies (Riesen et al. 2015). The running time was further reduced by defining ground costs for the assignment problem that are a tree metric (Kriege et al. 2019). This allows computing an optimal assignment in linear time by associating elements to the nodes of the tree and matching them in a bottom-up fashion. A tree metric gained from the Weisfeiler-Lehman refinement showed promising results.

2.1.2 Discussion

Various upper and lower bounds for the graph edit distance are known, some of which have been proposed for similarity search, while others are derived from algorithms for pairwise computation and are not directly suitable for fast searching in databases. Recently, an extensive study (Blumenthal et al. 2019) of different bounds confirmed, that there is a trade-off between computational efficiency and tightness. Lower bounds based on linear programming relaxations and the linear assignment problem were found to be most effective. However, the computation of such bounds requires solving an optimization problem and the combination with indices is non-trivial. Therefore, it has been proposed to compute graph embeddings optimized by graph neural networks, which reflect the graph edit distance, to make efficient index-based filtering possible (Qin et al. 2020). This and numerous other approaches (Li et al. 2019), (Bai et al. 2019) that use neural networks to approximate the similarity of graphs do not compute lower

or upper bounds on the graph edit distance and hence cannot be used to obtain exact results. Because of this, they are only suitable in situations in which incomplete answer sets are acceptable and are not in direct competition with exact approaches.

Recently, distance measures based on optimal assignments or, more generally, optimal transport (a.k.a. Wasserstein distance) have become increasingly popular for structured data. A method for approximate nearest neighbor search regarding the Wasserstein distance has been proposed recently (Backurs et al. 2020). Another line of work studies special cases, which allow vector space embeddings, e.g., in the domain of kernels for structured data (Kriege et al. 2016), (Le et al. 2019), (Kriege et al. 2019). On that basis we develop embeddings of novel assignment-based lower bounds for the graph edit distance, which are effective and allow index-accelerated similarity search, while guaranteeing exact results.

3 Preliminaries

We first give an overview of basic definitions concerning graph theory and database search. Then, we introduce tree metrics and the assignment problem, which play a major role in our new approach.

3.1 Graph theory

A *graph* $G = (V, E, \mu, \nu)$ consists of a set of vertices $V(G) = V$, a set of edges $E(G) = E \subseteq V \times V$, and labeling functions $\mu : V \to L$ and $\nu : E \to L$ for the vertices and edges. The labels L can be arbitrarily defined. We consider undirected graphs and denote an edge between u and v by uv. The *neighbors* of a vertex v are denoted by $N(v) = \{u \mid uv \in E(G)\}$ and the *degree* of v is $\delta(v) = |N(v)|$. The maximum degree of a graph G is $\delta(G) = \max_{v \in V} \delta(v)$ and we let $\Delta = \max_{G \in \mathrm{DB}} \delta(G)$ for the graph dataset DB.

A measure commonly used to describe the similarity of two graphs is the *graph edit distance*. An *edit operation* can be deleting or inserting an isolated vertex or an edge or relabeling any of the two. An *edit path* between graphs G and H is a sequence $(e_1, e_2, \ldots, e_k)$ of edit operations that transforms G into H. This means, that if we apply all operations in the edit path to G, we get a graph G' that is isomorphic to H, i.e., we can find a bijection $\xi : V(G') \to V(H)$, so that $\forall v \in V(G').\mu(v) = \mu(\xi(v)) \wedge \forall uv \in E(G').\nu(uv) = \nu(\xi(u)\xi(v))$. The graph edit distance is the cost of the (not necessarily unique) cheapest edit path.

Definition 1 (Graph Edit Distance (Riesen and Bunke 2009) Let c be a function assigning non-negative costs to edit operations. The *graph edit distance* between two graphs G and H is defined as

$$d_{\mathrm{ged}}(G, H) = \min \left\{ \sum_{i=1}^{k} c(e_i) \mid (e_1, \ldots, e_k) \in \Upsilon(G, H) \right\},$$

where $\Upsilon(G, H)$ denotes all possible edit paths from G to H.

Computation of the graph edit distance is NP-hard (Zeng et al. 2009). Hence, exact computation is possible only for small graphs. There are several heuristics, see Sect. 2, many of which are based on solving an assignment problem.

3.2 Optimal assignments and tree metrics

The assignment problem is a well-studied combinatorial optimization problem (Munkres 1957), (Burkard et al. 2012).

Definition 2 (Assignment Problem) Let A and B be two sets with $|A| = |B| = n$ and $c\colon A \times B \to \mathbb{R}$ a ground cost function. An *assignment* between A and B is a bijection $f\colon A \to B$. The *cost* of an assignment f is $c(f) = \sum_{a\in A} c(a, f(a))$. The *assignment problem* is to find an assignment with minimum cost.

For an assignment instance (A, B, c), we denote the cost of an optimal assignment by $d^c_{\mathrm{oa}}(A, B)$. The assignment problem can be solved in cubic running time using a suitable implementation of the Hungarian method (Munkres 1957), (Burkard et al. 2012). The running time can be improved when the cost function is restricted, e.g., to integral values from a bounded range (Duan and Su 2012). Of particular interest for our work is the requirement that the cost function is a tree metric, which allows to solve the assignment problem in linear time (Kriege et al. 2019) and relates the optimal cost to the Manhattan distance, see Sect. 4.1 for details. We summarize the concepts related to these distances.

Definition 3 (Metric) A *metric* d on X is a function $d\colon X \times X \to \mathbb{R}$ that satisfies the following properties for all $x, y, z \in X$: (1) $d(x, y) \geq 0$ (non-negativity), (2) $d(x, y) = 0 \iff x = y$ (identity of indiscernibles), (3) $d(x, y) = d(y, x)$ (symmetry), (4) $d(x, y) \leq d(x, z) + d(y, z)$ (triangle inequality).

The *Manhattan metric* (also *city-block* or ℓ_1 *distance*) is the metric function $d_m(\boldsymbol{x}, \boldsymbol{y}) = \|\boldsymbol{x} - \boldsymbol{y}\|_1 = \sum_{i=1}^{n} | x_i - y_i |$. A *tree* T is an acyclic, connected graph. To avoid confusion, we will call its vertices nodes. A tree with non-negative edge weights $w : E(T) \to \mathbb{R}_{\geq 0}$ yields a function $d_{T,w}(u, v) = \sum_{e\in P(u,v)} w(e)$ on $V(T)$, where $P(u, v)$ is the unique simple path from u to v in T.

Definition 4 (Tree Metric) A function $d : X \times X \to \mathbb{R}$ is a *tree metric* if there is a tree T with $X \subseteq V(T)$ and strictly positive real-valued edge weights w, such that $d(u, v) = d_{T,w}(u, v)$, for all $u, v \in X$.

Vice versa, every tree with strictly positive weights induces a tree metric on its nodes. Equivalently, a metric d is a tree metric iff $\forall v, w, x, y \in X.\ d(x, y) + d(v, w) \leq \max\{d(x, v) + d(y, w), d(x, w) + d(y, v)\}$ (Semple and Steel 2003). For such a tree with leaves X, a distinguished root, and the additional constraint that all paths from the root to a leaf have the same weighted length, the induced tree metric is an *ultrametric*. Equivalently, a metric d on X is an ultrametric if it satisfies the strong triangle inequality $\forall x, y, z \in X.\ d(x, y) \leq \max\{d(x, z), d(y, z)\}$ (Semple and Steel 2003). In the following, we also allow edge weight zero. Therefore, the distances induced by

a tree may violate property (2) of Definition 3 and are therefore *pseudometrics*. For the sake of simplicity, we still use the terms tree metric and ultrametric.

We consider the assignment problem, where the cost function $c\colon A \times B \to \mathbb{R}$ is a tree metric $d : X \times X \to \mathbb{R}$. To formalize the link between these two functions and, hence, Definitions 2 and 4, we introduce the map $\varrho\colon A \cup B \to X$. Given a tree metric specified by the tree T with weights w and the map ϱ, the cost for assigning an object $a \in A$ to an object $b \in B$ is defined as $c(a, b) = d_{T,w}(\varrho(a), \varrho(b))$. Note that ϱ is not required to be injective. Therefore, c may be a pseudometric even for trees with strictly positive weights. The input size of the assignment problem according to Definition 2 typically is quadratic in n as c is given by an $n \times n$ matrix. If c is a tree metric, it can be compactly represented by the tree T with weights w having a total size linear in n.

3.3 Searching in databases

Databases can store data in order to retrieve, insert or change it efficiently. Regarding data analysis, retrieval (search) is usually the crucial operation on databases, because it will be performed much more often than updates. We focus on two types of similarity queries when searching a database DB, the first of which is the *range query* for a radius r.

Definition 5 (Range Query) Given a query object q and a threshold r, determine $\text{range}(q, r) = \{o \in \text{DB} \mid d(o, q) \leq r\}$.

A range query finds all objects with a distance no more than the specified range threshold r to the query object q. If the distance d is expensive to compute, it makes sense to use the so-called filter-verification principle. In this approach different lower and upper bounds are used to filter out a hopefully large portion of the database. A function d' is a *lower bound* on d if $d'(x, y) \leq d(x, y)$, and an *upper bound* if $d'(x, y) \geq d(x, y)$ for all $x, y \in X$. Clearly, objects where one of the lower bounds is greater than r can be dismissed since the exact distance would be even greater. Objects, where an upper bound is at most r can be added to the result immediately. Only the remaining objects need to be verified by computing the exact distance.

The second type of query considered here is the *nearest neighbor query*, which returns the objects that are closest to the query object.

Definition 6 (k-Nearest Neighbor Query, $k$nn Query) Given a query object q and a parameter k, determine the smallest set $\text{NN}(q, k) \subseteq \text{DB}$, so that $|\text{NN}(q, k)| \geq k$ and $\forall o \in \text{NN}(q, k), \forall o' \in \text{DB} \setminus \text{NN}(q, k) : \; d(o, q) < d(o', q)$.

In conjunction with range queries, it is preferable to return all the objects with a distance, that does not exceed the distance to the kth neighbor, which may be more than k objects when tied. That yields an equivalency of the results of $k$nn queries and range queries, i.e., we have $\text{range}(q, r) = \text{NN}(q, |\text{range}(q, r)|)$ and $\text{NN}(q, k) = \text{range}(q, r_k)$, where r_k is the maximum distance in $\text{NN}(q, k)$.

The optimal multi-step k-nearest neighbor search algorithm (Seidl and Kriegel 1998) minimizes the number of candidates verified by using an incremental neighbor

search, which returns the objects in ascending order regarding the lower bound. As new candidates are discovered, their exact distance is computed. The current kth smallest exact distance is used as a bound on the incremental search: once we have found at least k objects with an exact distance smaller than the lower bound of all remaining objects, the result is complete. This approach is optimal in the sense that none of the exact distance computations could have been avoided (Seidl and Kriegel 1998).

4 EmbAssi: embedding assignment costs for graph edit distance lower bounds

We define lower bounds for the graph edit distance obtained from optimal assignments regarding tree metric costs. These bounds are embedded in ℓ_1 space and used for index-accelerated filtering. In Sect. 4.1 we describe the general technique for embedding optimal assignment costs for tree metric ground costs based on (Kriege et al. 2019). In Sect. 4.2 we propose several embeddable lower bounds for the graph edit distance derived from such assignment problems. These are suitable for graphs with discrete labels and uniform edit costs and are generalized to non-uniform edit costs. In Sects. 5.1 and 5.2 we show how to use these bounds for both range and k-nearest neighbor queries and discuss optimization details.

4.1 Embedding assignment costs

Let (A, B, c) be an assignment problem, where the cost function c is a tree metric defined by the tree T with weights w. The cost of an optimal assignment is equal to the Manhattan distance between vectors derived from the sets A and B using the tree T and weights w (Kriege et al. 2019). Recall that the cost of an assignment is the sum of the costs of all matched pairs. A matched pair (a, b) contributes the cost defined by the weight of the edges on the unique path between the nodes $\varrho(a)$ and $\varrho(b)$ in T. Hence, the total cost can be obtained from the number of times the edges occur on such paths.

Let $S_{\overleftarrow{uv}}$ correspond to the number of elements of a set S, that are associated by the mapping ϱ with nodes in the subtree of T containing u, when the edge uv is deleted (see Fig. 1). It was shown in (Kriege et al. 2019) that an optimal assignment has cost

$$d_{\mathrm{oa}}^{c}(A, B) = \sum\nolimits_{uv \in E(T)} \mid A_{\overleftarrow{uv}} - B_{\overleftarrow{uv}} \mid \cdot w(uv). \tag{1}$$

Note that the roles of u and v are interchangeable and we indicate the choice by directing the edge accordingly. Although this does not affect the assignment cost when applied consistently, there are subtle technical consequences, which we discuss for concrete tree metrics in Sect. 5.1. Using T and w we can map sets to vectors having a component for every edge of T defined as $\Phi_c(S) = [S_{\overleftarrow{uv}} \cdot w(uv)]_{uv \in E(T)}$. From Eq. 1 it directly follows that the optimal assignment cost are

$$d_{\mathrm{oa}}^{c}(A, B) = \|\Phi_c(A) - \Phi_c(B)\|_1.$$

This embedding of the optimal assignment cost into ℓ_1 space is used in the following to obtain assignment-based lower bounds on the graph edit distance.

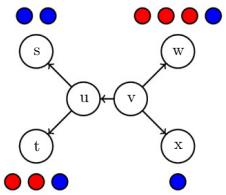
4.2 Embeddable lower bounds

Several lower bounds on the graph edit distance can be obtained from optimal assignments (Blumenthal et al. 2019). However, these typically do not use a tree metric cost function, which complicates the embedding of assignment costs. In (Kriege et al. 2019) two tree metrics, one based on Weisfeiler-Lehman refinement for graphs with discrete labels and one using clustering for attributed graphs, were introduced. These, however, both do not yield a lower bound. We develop new lower bounds on the graph edit distance from optimal assignment instances, which have a tree metric ground cost function. Most similarity search techniques for the graph edit distance assume a uniform cost model, where every edit operation has the same cost. We also support variable cost functions and discuss choices that are supported by our approach. We use c_v/c_e to denote the costs of inserting or deleting vertices/edges and c_{vl}/c_{el} for the costs of changing the respective label.

4.2.1 Vertex label lower bounds

A natural method for defining a lower bound on the graph edit distance is to just take the labels into account ignoring the graph structure. We first discuss the case of uniform cost for changing a label, which is common for discrete labels. Then, non-uniform costs are considered.

Uniform Cost Functions Clearly, each vertex of one graph, that cannot be assigned to a vertex of the other graph with the same label has to be either deleted or relabeled. The idea leads to a particularly simple assignment instance when we assume fixed costs c_{vl} and c_v. Let G and H be two graphs with n respectively m vertices. Following the common approach to obtain an assignment instance (Riesen and Bunke 2009), we



(a) Tree T representing metric c

	$\overleftarrow{su}$	$\overleftarrow{tu}$	$\overleftarrow{uv}$	$\overleftarrow{wv}$	$\overleftarrow{xv}$
$\Phi_c(A)$	0	2	2	3	0
$\Phi_c(B)$	2	1	3	1	1

(b) Embeddings

Fig. 1 **a** An assignment problem (A, B, c) with a tree T representing the metric c. The elements of A are denoted by red circle and the elements of B by blue circle and associated to the nodes of T by ϱ as depicted. All edges have weight 1. **b** Embedding of A and B regarding T. The entry $\overleftarrow{uv}$ for the set B counts the total number of blue circle elements associated with the nodes s, t and u as indicated by the direction of the edge uv in the tree. The assignment cost is $d^c_{\mathrm{oa}}(A, B) = 7$

extend G by m and H by n dummy nodes denoted by ϵ.[1] We consider the following assignment problem.

Definition 7 (Label Assignment) The *label assignment* instance for G and H is given by $(V(G), V(H), c_{\text{llb}})$, where the ground cost function is

$$c_{\text{llb}}(u, v) = \begin{cases} 0 & \text{if } \mu(u) = \mu(v) \text{ or } u = v = \epsilon \\ c_{vl} & \text{if } \mu(u) \neq \mu(v) \\ c_v & \text{if either } u = \epsilon \text{ or } v = \epsilon. \end{cases}$$

We define $LLB(G, H) = d_{\text{oa}}^{c_{\text{llb}}}(V(G), V(H))$ and show that it provides a lower bound on the graph edit distance.

Proposition 1 (Label lower bound) *For any two graphs G and H, we have $LLB(G, H) \leq GED(G, H)$.*

Proof Every assignment directly induces a set of edit operations, which can be arranged to form an edit path. Vice versa, every edit path can be represented by an assignment (Riesen and Bunke 2009), (Blumenthal et al. 2019). Let $\boldsymbol{e}$ be a minimum cost edit path. We construct an assignment f from the vertex operations in $\boldsymbol{e}$, where the deletion of v is represented by $(v, \epsilon) \in f$, insertion by $(\epsilon, v) \in f$, and relabeling of the vertex u with the label of v by $(u, v) \in f$, where $u, v \neq \epsilon$. We have $c(\boldsymbol{e}) = Z_v + Z_e$, where Z_v and Z_e are the costs of vertex and edge edit operations, respectively. According to the definition of c_{llb} and the construction of f we have $Z_v = c_{\text{llb}}(f)$. An optimal assignment o satisfies $c_{\text{llb}}(o) \leq c_{\text{llb}}(f)$ and $LLB(G, H) = c_{\text{llb}}(o) \leq c_{\text{llb}}(f) \leq c(\boldsymbol{e}) = GED(G, H)$ follows, since $Z_e \geq 0$. □

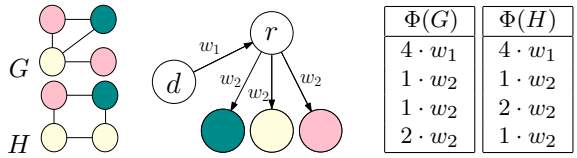
To obtain embeddings, we investigate for which choices of edit costs the ground cost function c_{llb} is a tree metric.

Proposition 2 (LLB tree metric) *The ground cost function c_{llb} is a tree metric if and only if $c_{vl} \leq 2c_v$.*

Proof First we assume $c_{vl} \leq 2c_v$ and define a tree T with a central node r having a neighbor for every label $l \in L$ and a neighbor d. Let $w(rl) = \frac{1}{2}c_{vl}$ for all $l \in L$ and $w(rd) = c_v - \frac{1}{2}c_{vl}$, cf. Figure 2b. The assumption guarantees that all weights are non-negative. We consider the map $\varrho(v) = \mu(v)$ for $v \neq \epsilon$ and $\varrho(\epsilon) = d$. We observe that $c_{\text{llb}}(u, v) = d_{T,w}(\varrho(u), \varrho(v))$ by verifying the three cases.

The reverse direction is proven by contradiction. Assume $c_{vl} > 2c_v$ and c_{llb} a tree metric. Let u and v be two vertices with $\mu(u) \neq \mu(v)$, then $c_{\text{llb}}(u, v) = c_{vl}$ and $c_{\text{llb}}(u, \epsilon) = c_{\text{llb}}(\epsilon, v) = c_v$. Therefore, $c_{\text{llb}}(u, v) > c_{\text{llb}}(u, \epsilon) + c_{\text{llb}}(\epsilon, v)$ contradicting the triangle inequality, Definition 3, (4). Thus, c_{llb} is not a metric and, in particular, not a tree metric contradicting the assumption. □

[1] One can also add only a single dummy node and modify the definition of an assignment (Blumenthal et al. 2019). However, this makes no difference for our technique, which embeds the assignment costs. We discuss how dummy vertices can be avoided in Sect. 5.1.



$\Phi(G)$	$\Phi(H)$
$4 \cdot w_1$	$4 \cdot w_1$
$1 \cdot w_2$	$1 \cdot w_2$
$1 \cdot w_2$	$2 \cdot w_2$
$2 \cdot w_2$	$1 \cdot w_2$

(a) Graphs **(b)** Weighted tree **(c)** Embeddings

Fig. 2 Two graphs G and H **a**, the weighted tree representing the ground cost function c_{llb} **b**, and the derived embeddings **c**. The weights are $w_1 = c_v - \frac{1}{2}c_{vl}$ and $w_2 = \frac{1}{2}c_{vl}$. The entries of the vectors correspond to the edges of the tree, from left to right, arrows indicate the direction used when counting elements

The requirement $c_{vl} \leq 2c_v$ states that relabeling a vertex is at most as expensive as deleting and inserting it with the correct label. This is generally reasonable and not a severe limitation. Because the proof is constructive, it allows us to represent c_{llb} by a weighted tree, from which we can compute the graph embedding representing the assignment costs following the approach described in Sect. 4.1.

Figure 2 illustrates the embedding of the label lower bound for an example. The tree representing the cost function is shown in Fig. 2b. The weight of the edge from the dummy node to the root is chosen, such that the path length from a label to the dummy node is c_v. Figure 2c shows the vectors Φ of the two example graphs, which allows obtaining $LLB(G, H) = \|\Phi(G) - \Phi(H)\|_1 = c_{vl}$ as the Manhattan distance.

Non-Uniform Cost Functions We have discussed the case where changing one label into another has a fixed cost of c_{vl}. In general, the cost for this may depend on the two labels involved, i.e., we assume that a cost function $c_{vl} \colon L \times L \to \mathbb{R}_{\geq 0}$ is given. Two common scenarios can be distinguished: First, L is a (small) finite set of labels that are similar to varying degrees. An example are molecular graphs, where the costs are defined based on vertex labels encoding their pharmacophore properties (Garcia-Hernandez et al. 2019). Second, L is infinite (or very large), e.g., vertices are annotated with coordinates in $\mathbb{R}^2$ and the cost is defined as the Euclidean distance. We propose a general method and then discuss its applicability to both scenarios.

We can extend the tree defining the metric used in the above paragraph to allow for more fine-grained vertex relabel costs. To this end, an arbitrary ultrametric tree on the labels L is defined, where the node d representing deletions is added to its root r. Recall that in an ultrametric tree the lengths of all paths from the root to a leaf are equal to, say, u. We define the weight of the edge between r and d as $c_v - u$ and observe that $c_v \geq u$ is required to obtain a valid tree metric in analogy to the proof of Proposition 2.

To obtain an ultrametric tree that reflects the given edit cost function c_{vl}, we employ hierarchical clustering. To guarantee that the assignment costs are a lower bound on the graph edit distance, it is crucial that interpreting the hierarchy as an ultrametric tree will underestimate the real edit costs. For optimal results, we would like to obtain a tight lower bound. We formalize the requirements. Let $c_{vl} \colon L \times L \to \mathbb{R}_{\geq 0}$ be the given cost function and $d_{hc} \colon L \times L \to \mathbb{R}_{\geq 0}$ the ultrametric induced by hierarchical clustering of L with cost function c_{vl}. Let $U^{-}(c_{vl})$ be the set of all ultrametrics that are lower bounds on c_{vl}. There is a unique ultrametric $d^* \in U^{-}(c_{vl})$ defined as $d^*(l_1, l_2) = \sup_{d \in U^{-}(c_{vl})}\{d(l_1, l_2)\}$ for all $l_1, l_2 \in L$ (Bock 1974). This d^* is an

upper bound on all ultrametrics in $U^{-}(c_{vl})$, a lower bound on c_{vl} and called the *subdominant* ultrametric to c_{vl}. The subdominant ultrametric is generated by single-linkage hierarchical clustering (Bock 1974), which therefore is, in this respect, optimal for our purpose. In particular, it reconstructs an ultrametric tree if the original costs are ultrametric. Moreover, single-linkage clustering can be implemented with running time $O(|L|^2)$ (Sibson 1973).

For a finite set of labels L, our method is a viable solution if the edit cost function c_{vl} is close to an ultrametric and L is small. If L is infinite, we need to approximate it with a finite set through quantization or space partitioning. The realization of such an approach preserving the lower bound property depends on the specific application and is hence not further explored here.

4.2.2 Degree lower bound

The *LLB* does not take the graph structure into account. We now introduce the degree lower bound, which focuses on how many edges have to be inserted or deleted at the minimum. When deleting or inserting vertices, all of the adjacent edges have to be deleted or inserted as well. If two vertices with differing degrees are assigned to one another, again edges have to be deleted or inserted accordingly. As in Sect. 4.2.1, we extend the graphs G and H by dummy nodes ϵ and define an assignment problem.

Definition 8 (Degree Assignment) The *degree assignment* instance for G and H is given by $(V(G), V(H), c_{\text{dlb}})$, where the ground cost function is $c_{\text{dlb}}(u, v) = \frac{1}{2}c_e \mid \delta(u) - \delta(v) \mid$ with $\delta(\epsilon) := 0$ for the dummy nodes.

We define $DLB(G, H) = d_{\text{oa}}^{c_{\text{dlb}}}(V(G), V(H))$, and show that it is a lower bound.

Proposition 3 (Degree lower bound) *For any two graphs G and H, we have $DLB(G, H) \leq GED(G, H)$.*

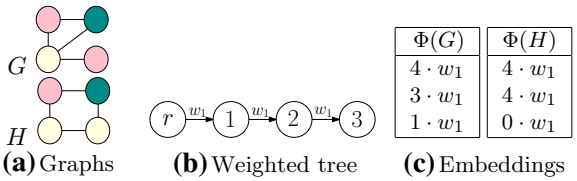
Proof Using the same arguments as in the proof of Proposition 2, let $\boldsymbol{e}$ be a minimum cost edit path and f an assignment that induces $\boldsymbol{e}$. We divide the costs $c(\boldsymbol{e}) = Z_v + Z_e$ into costs Z_v and Z_e of vertex and edge edit operations. For the matched vertices v and $f(v)$ at least $\mid \delta(v) - \delta(f(v)) \mid$ edges must be deleted or inserted to balance the degrees; in case of insertion and deletion all adjacent edges must be inserted or deleted. Since each edge edit operation increases or decreases the degree of its two endpoints by one, the sum of these costs over all vertices must be divided by two and $c_{\text{dlb}}(f) = Z_e$ follows. □

To obtain an embedding, we show that c_{dlb} is a tree metric.

Proposition 4 (DLB tree metric) *The ground cost function c_{dlb} is a tree metric.*

Proof To prove that c_{dlb} is a tree metric, we construct a tree T with edge weights w and a map ϱ, so that $c_{\text{dlb}}(u, v) = d_{T,w}(\varrho(u), \varrho(v))$. Let T have nodes $V(T) = \{r = 0, 1, \ldots, \Delta\}$ and edges $E(T) = \{ij \mid j = i{+}1\}$ with weight $w_1 = \frac{1}{2}c_e$. Since c_e cannot be negative, all edge weights are non-negative. We consider the map

$$\varrho(u) = \begin{cases} r & \text{if } u = \epsilon \text{ or } \delta(u) = 0 \\ \delta(u) & \text{otherwise.} \end{cases}$$



(a) Graphs **(b)** Weighted tree **(c)** Embeddings

Fig. 3 Two graphs G and H **a**, the weighted tree representing the cost function c_{dlb} **b**, and the derived embeddings **c**

It can easily be seen, that $c_{\text{dlb}}(u, v) = d_{T,w}(\varrho(u), \varrho(v))$ by verifying the path lengths in the tree. □

The proof gives a concept to construct a tree representing the *DLB* cost function. As there is no difference between a vertex with degree 0 and a dummy vertex, they can both be assigned to the root node r. Note, that the edge labels are not taken into account by this lower bound and edge insertion and deletion are not distinguished. Figure 3 illustrates the embedding of the degree lower bound, which yields $DLB(G, H) = c_e$ for the running example.

4.2.3 Combined lower bound

We can combine *LLB* and *DLB* to improve the approximation.

Definition 9 (CLB) The *combined lower bound* between G and H is defined as $CLB(G, H) = LLB(G, H) + DLB(G, H)$.

We show, that *CLB* is a lower bound on the graph edit distance. Note that this lower bound is based on the two assignments given by *LLB* and *DLB*, which are not necessarily equal.

Lemma 1 *Let c_1, c_2 and c be ground cost functions on X and $c(x, y) = c_1(x, y) + c_2(x, y)$ for all $x, y \in X$. Then for any $A, B \subseteq X$, $|A| = |B| = n$, the inequality $d^{c_1}_{\text{oa}}(A, B) + d^{c_2}_{\text{oa}}(A, B) \leq d^{c}_{\text{oa}}(A, B)$ holds.*

Proof Let o_1, o_2 and o be optimal assignments between A and B regarding the ground costs c_1, c_2 and c, respectively. Due to the optimality we have $c_1(o_1) \leq c_1(o)$ and $c_2(o_2) \leq c_2(o)$. Hence, $d^{c_1}_{\text{oa}}(A, B) + d^{c_2}_{\text{oa}}(A, B) = c_1(o_1) + c_2(o_2) \leq c_1(o) + c_2(o) = c(o) = d^{c}_{\text{oa}}(A, B)$. □

Proposition 5 *(Combined lower bound) For any two graphs G and H, we have $CLB(G, H) \leq GED(G, H)$.*

Proof Let $\boldsymbol{e}$ be a minimum cost edit path and f an assignment that induces $\boldsymbol{e}$. We divide the costs $c(\boldsymbol{e}) = Z_v + Z_e$ into costs Z_v and Z_e of vertex and edge edit operations. From the proof of Propositions 1 and 3 we know that $Z_v \geq c_{\text{llb}}(f)$ and $Z_e \geq c_{\text{dlb}}(f)$ and, hence, $c(\boldsymbol{e}) \geq c_{\text{llb}}(f) + c_{\text{dlb}}(f)$. Application of Lemma 1 yields $CLB(G, H) = c_{\text{llb}}(f_1) + c_{\text{dlb}}(f_2) \leq c_{\text{llb}}(f) + c_{\text{dlb}}(f) \leq c(\boldsymbol{e}) = GED(G, H)$, where f_1 and f_2 are optimal assignments regarding c_{llb} and c_{dlb}. □

This lower bound is at least as tight as the ones it consists of and, therefore, most promising. The combined lower bound is embedded by concatenating the vectors for *LLB* and *DLB*.

4.3 Analysis

We provide a theoretical comparison of our proposed bounds to existing lower bounds and also give details on the time complexity of our approach.

4.3.1 Comparison with existing bounds

We relate the *CLB* to two well-known lower bounds when applied to graphs with vertex labels. The *simple label filter* (*SLF*) is the intersection of vertex and edge label multisets, i.e., $SLF(G, H) = |L_V(G) \cap L_V(H)| + ||E(G)| - |E(H)||$ in our case, where L_V denotes the vertex label multiset of a graph. Although simple, this bound is often found to be selective (Kim et al. 2019) and, therefore, widely-used (Zhao et al. 2012), (Zhao et al. 2013). A very effective bound according to (Blumenthal et al. 2019) is *BranchLB* based on general optimal assignments. Several variants have been proposed (Riesen and Bunke 2009), (Zheng et al. 2015), (Blumenthal et al. 2019) with at least cubic worst-case time complexity. In our case, *BranchLB* is the cost of the optimal assignment regarding the ground costs $c_{\text{branch}}(u, v) = c_{\text{llb}}(u, v) + c_{\text{dlb}}(u, v)$. Note that c_{branch} in general is not a tree metric. *SLF* assumes $c_v = c_{vl} = c_e = 1$ and we consider this setting although *CLB* and *BranchLB* are more general. Using counting arguments and Lemma 1 we obtain the following relation:

Proposition 6 *For any two vertex-labeled graphs G and H,* $SLF(G, H) \leq CLB(G, H) \leq BranchLB(G, H) \leq GED(G, H)$.

Experimentally we show in Sect. 6 that our combined lower bound is close to *BranchLB* for a wide-range of real-world datasets, but is computed several orders of magnitude faster and allows indexing. This makes it ideally suitable for fast pre-filtering and search.

4.3.2 Time complexity

We first consider the time required for generating the vector $\Phi_c(S)$ for a set S and tree T defining the ground cost function c.

Proposition 7 *Given a set S and a weighted tree T representing the ground cost function c, the vector* $\Phi_c(S)$ *can be computed in* $O(|V(T)| + |S|)$ *time.*

Proof We first associate the elements of S with the nodes of T via the map ϱ and then traverse T starting from the leaves progressively moving towards the center. The order guarantees that when the node u is visited, exactly one of its neighbors, say v, has not yet been visited. Then $S_{\overleftarrow{uv}}$ can be obtained as $\sum_{w \in N(u) \setminus \{v\}} S_{\overleftarrow{wu}}$ from the values computed previously. The tree traversal and computation of $S_{\overleftarrow{uv}}$ for all $uv \in E(T)$ takes $O(|V(T)|)$ total time. Together with the time for processing the set S we obtain $O(|V(T)| + |S|)$ time. □

The time complexity of the different bounds depends on the size of the tree representing the metric and the size of the graphs.

Proposition 8 *The bounds c_{llb}, c_{dlb} and c_{clb} for two graphs G and H can be computed in $O(|V(G)| + |V(H)|)$ time.*

Proof First the tree T defining the metric is computed. For the different tree metrics, the trees sizes are linear in the number of nodes of the two graphs G and H: For c_{llb} the tree (denoted T_{llb}) has size $|L_v(G) \cup L_v(H)| + 2$, where $L_v(G)$ denotes the set of vertex labels occurring in G. The tree consists of a node for each vertex label plus a dummy and a central node. In the worst case, where all labels occur only once, the tree is of size $|V(G)| + |V(H)| + 2$. For c_{dlb}, we have $|V(T_{\text{dlb}})| = \max(\delta(G), \delta(H)) + 1$, since there is a node for each vertex degree, up to the maximum degree (including degree 0). As shown in Proposition 7, the vector $\Phi_c(G)$ can then be computed in time $O(|V(T)| + |V(G)|)$. For c_{clb} we concatenate the vectors of c_{llb} and c_{dlb}. The Manhattan distance between two vectors is computed in time linear in the number of components, which is $O(|V(T)|)$. Thus, the total running time for computing any of the bounds is in $O(|V(G)| + |V(H)|)$. □

The bound *SLF* also has a linear time complexity while *BranchLB* requires $O(n^2\Delta^3 + n^3)$ time for graphs with n vertices and maximum degree Δ (Blumenthal et al. 2019). Our new approach matches the running time of *SLF* but in most cases yields tighter bounds, cf. Proposition 6 and our experimental evaluation in Sect. 6. Hence, it provides a favorable trade-off between efficiency and quality and at the same time can conveniently be combined with indices.

5 EmbAssi for graph similarity search

We use the proposed lower bounds for similarity search by computing embeddings for all the graphs in the database in a preprocessing step. Given a query graph, we

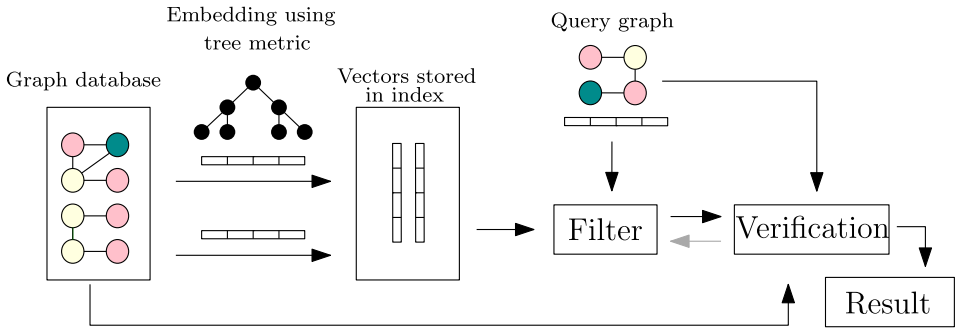


Fig. 4 Overview of our pipeline for graph similarity search. In a preprocessing step the embeddings of all database graphs under the specified tree metric are computed and stored in an index. Then similarity search queries are answered by computing the embedding of the query graph and filtering regarding the Manhattan distance. In k-nearest neighbor search, the information gained from refining candidates is used to reduce the search range. This is indicated with a gray arrow

Algorithm 1 Construction for *LLB*

procedure CONSTRUCTINDEX(set of graphs DB)
  $T \leftarrow$ new tree with root ϵ and child r
  $w((\epsilon, r)) \leftarrow c_v - \frac{1}{2}c_{vl}$ ▷ Set edge weight
  for all $G \in DB$ **do** ▷ Construct tree metric *LLB*
    for all $v \in V(G)$ **do**
      if $\mu(v) \notin T$ **then**
        add node $\mu(v)$ as child of r with $w((r, \mu(v))) = \frac{1}{2}c_{vl}$
        $\varrho(v) \leftarrow \mu(v)$
      end if
    end for
  end for
  for all $G \in DB$ **do** ▷ Compute embedding under tree metric
    $\Phi(G) \leftarrow$ COMPUTEVECTOR(G, T)
  end for
  Create index($\{\Phi(G) \mid G \in DB\}$, Manhattan distance)
end procedure

Algorithm 2 Vector Computation

procedure COMPUTEVECTOR(graph G, tree T)
  $\Phi(G) \leftarrow$ sparse vector
  $S \leftarrow$ relevant subtree($V(G), T$) ▷ $\forall v \in V(G) : \varrho(v)$ and path to root
  $count[n] \leftarrow 0, \forall n \in V(S)$
  for all $v \in V(G)$ **do**
    $count[\varrho(v)] \leftarrow count[\varrho(v)] + 1$ ▷ Count vertices at leaves
  end for
  $leaves \leftarrow$ leaves of S
  while $l \leftarrow leaves$.dequeue() **and** $l \neq$ root of S **do**
    $\Phi(G)[(l, p)] \leftarrow w((l, p)) \cdot count[l]$
    $count[p] = count[p] + count[l]$ ▷ Propagate vertex count to parent
    delete l from S
    if $\delta(p) \leq 1$ **then** ▷ p is now a leaf itself
      $leaves$.enqueue(p)
    end if
  end while
  return $\Phi(G)$
end procedure

compute its embedding and realize filtering utilizing indices regarding the Manhattan distance. The approach is illustrated in Fig. 4. Algorithm 1 shows how the preprocessing is done, exemplary for *LLB*: We construct the tree metric based on the labels and associate the vertices with the leaves of the tree. Then for each graph the embedding is computed using Algorithm 2.

Several technical details must be considered. The choice of how to direct the edges has a huge impact on the resulting vectors and of course using a suitable index is also important. In the following, we briefly discuss our choices and explain how similarity search queries can be answered.

5.1 Index construction

We compute the vectors for all graphs in the database and store them in an index to accelerate queries. When defining the bounds, we considered the pairwise comparison of two graphs and added dummy vertices to obtain graphs of the same size. We have chosen the direction of edges in the trees representing the metrics carefully, cf. Section 4.1, to generate consistent embeddings for the entire database. By rooting the trees at the node $\varrho(\epsilon)$ representing the dummy vertices (see Algorithm 1) and directing all edges towards the leaves the dummy vertices are not counted in any entry of the vectors, see Fig. 2 and 3. Moreover, this choice often leads to sparse vectors, e.g., for the *LLB*, where every entry just counts the number of vertices with one specific label. Labels that only appear in a small fraction of the graphs in the database then lead to zero-entries in the vectors of the other graphs and sparse data structures become beneficial. Moreover, this simplifies to dynamically add new vertex labels without requiring to update all existing vectors in the database. Using sparse vector representations, Algorithm 2 can be implemented in time $O(|V(G)|)$ by considering only the relevant part of T. This is the subtree S formed by the nodes of T to which vertices of G are assigned to via ϱ, and the nodes on the path to the root from these nodes. The subtree S is processed in a bottom-up fashion computing a non-zero component of $\Phi(G)$ in each step. Note that S can be maintained and modified with low overhead using flags to indicate whether a node of T is contained in S.

The choice of a suitable index is crucial for the performance of our approach. We chose to use the cover tree (Beygelzimer et al. 2006) because our data is too high-dimensional for the popular k-d-tree, and our vectors have many zeros and discrete values. The cover tree is a good choice for an in-memory index because of its lightweight construction, low memory requirements, and good metric pruning properties. It is usually superior to the k-d-tree or R-tree if the data stored is high-dimensional but still has a small doubling dimension.

5.2 Queries

For similarity search, we compute the embedding of the query graph and use the index for similarity search regarding Manhattan distance. The index takes responsibility to disregard parts of the database that are too far away from the query object. In k-nearest neighbor search, we use the optimal multi-step k-nearest neighbor search (Seidl and Kriegel 1998) as described in Sect. 3.3 to stop the search as early as possible and compute the minimum necessary number of exact graph edit distances. Our lower bounds are especially useful for this because it is well understood how to index data for ranking by Manhattan distance. Further exact distance computations (in particular for range queries) can be avoided by checking additional bounds similar to *Inves* (Kim et al. 2019) or *BSS_GED* (Chen et al. 2019) prior to an exact distance computation.

A tighter (but more expensive) lower bound produces fewer candidates, while in some applications (such as DBSCAN clustering), where the exact distance is not necessary to have, an upper bound can identify true positives efficiently.

6 Experimental evaluation

In this section, we compare *EmbAssi* to state-of-the-art approaches regarding efficiency and approximation quality in range and k-nearest neighbor queries. We investigate the speed-up of existing filter-verification pipelines when *EmbAssi* is used in a pre-filtering step. Specifically, we address the following research questions:

- **Q1** How tight are our lower bounds compared to the state-of-the-art? How do our bounds perform when taking the trade-off between bound quality and runtime into account?
- **Q2** Can *EmbAssi* compete with state-of-the-art methods in terms of runtime and selectivity? Is *CLB* a suitable lower bound to provide initial candidates for range queries?
- **Q3** Can *EmbAssi* perform similarity search on datasets with a million graphs or more?
- **Q4** Can k-nearest neighbor queries be answered efficiently?

6.1 Setup

This section gives an overview of the datasets, the methods, and their configuration used in the experimental comparison.

Methods and Distance Functions We compare *EmbAssi* to *GSim* (Zhao et al. 2012) and *MLIndex* (Liang and Zhao 2017), which are representative methods for similarity search based on overlapping substructures and graph partitioning. *MLIndex* is considered as state-of-the-art (Qin et al. 2020), although we observed that *GSim* often performs much better. We also compare to *CStar* (Zeng et al. 2009) and *Branch* (Blumenthal et al. 2019), which provide both upper and lower bounds on the graph edit distance, but are not accelerated with indices. Furthermore, we compare to the exact graph edit distance *BLP* (Lerouge et al. 2017), *BSS_GED* (Chen et al. 2019), and the approximations *LinD* (Kriege et al. 2019), *BLPlb* (Blumenthal et al. 2019) and *BeamS* (Neuhaus et al. 2006) regarding the approximation of the *GED*. The costs of all edit operations were set to one because some of the comparison methods only support uniform costs. For *BeamD*, we used a maximum list size of 100. For *LinD*, the tree was generated using the Weisfeiler-Lehman algorithm with one refinement iteration. For *GSim*, we used all provided filters and $q = 3$. For *MLIndex*, the default settings of the authors' implementation were used.

The bounds computed by *CStar* can also be used separately and will be referred to as *CStarLB*, *CStarUB*, and *CStarUBRef*, which is obtained by improving an edit path using local search. *SLF* and *BranchLB* are the lower bounds discussed in Sect. 4.2.3 and were implemented following the description in (Kim et al. 2019) and (Blumenthal et al. 2019), respectively. *BranchLB* and the upper bound gained from the edit path induced by it (*BranchUB*) are referred to as *Branch* when applied together. Table 2 gives an overview of the distance functions compared in the experiments including those known from literature as well as those proposed here for use with *EmbAssi*. The graphs and their respective vectors are indexed using the cover tree (Beygelzimer et al. 2006) implementation of the ELKI framework (Schubert and Zimek 2019).

Table 2 Distance functions compared in the experiments

Name	Description	Bound	Source
BLP	Exact graph edit distance	exact	(Lerouge et al. 2017)
BSS_GED	Efficient ged computation/verification	exact	(Chen et al. 2019)
BeamD	Approximation using BeamSearch	upper	(Neuhaus et al. 2006)
LinD	Optimal assignments with WL	upper	(Kriege et al. 2019)
BranchUB	Cost of edit path gained from BranchLB	upper	(Blumenthal et al. 2019)
CStarUB	Optimal assignments with stars	upper	(Zeng et al. 2009)
CStarUBRef	Refined Version of *CStarUB*	upper	(Zeng et al. 2009)
CStarLB	Mapping distance between stars	lower	(Zeng et al. 2009)
SLF	Min. label changes	lower	(Kim et al. 2019)
BranchLB	Modification of *BP*	lower	(Blumenthal et al. 2019)
BLPlb	Relaxation of *BLP*	lower	(Blumenthal et al. 2019)
New distance functions based on tree metrics			
LLB	Min. vertex label changes	lower	new
DLB	Min. edge insertion/deletions	lower	new
CLB	*LLB* and *DLB* combined	lower	new

Table 3 Datasets with discrete vertex labels and their statistics Morris et al. (2020). *ChEMBL* (*chembl_27*, Gaulton et al. (2016)) contains small molecules, *Protein Com* contains protein complex graphs Stöcker et al. (2019)

Name	|Graphs|	avg $\|V\|$	avg $\|E\|$	avg $\delta(v)$	$\|L\|$
KKI	83	26.96	48.42	3.59±2.58	190
MCF-7	27770	26.08	28.29	2.16±0.76	23
MUTAG	188	17.93	19.79	2.20±0.74	7
NCI1	4110	29.04	31.61	2.16±0.78	21
PTC_FM	349	14.11	14.48	2.05±0.81	18
Protein Com	1455324	9.98	8.98	1.80±3.03	717
ChEMBL	1941411	30.18	32.71	2.15±0.74	31

Datasets We tested all methods on a wide range of real-world datasets with different characteristics, see Table 3. The datasets have discrete vertex labels. Edge labels and attributes, if present, were removed prior to the experiments since not all methods support them. Also, since *MLIndex* and *GSim* do not work for disconnected graphs, only their largest connected components were used.

6.2 Results

In the following, we report on our experimental results and discuss the different research questions.

Q1: Bound Quality and Runtime Accuracy is crucial to obtain effective filters for similarity search. We investigate how tight the proposed lower bounds on the graph edit distance are. Figure 5 shows the average relative approximation error $\frac{|\text{GED}-d_{\text{approx}}|}{\text{GED}}$

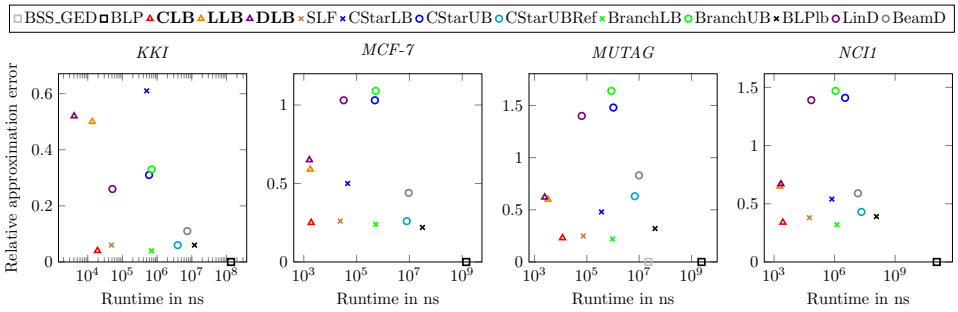


Fig. 5 Comparison of several different approximations regarding their relative approximation error and runtime. □: exact approach, ○: upper bound, ×: existing lower bound, △: newly proposed lower bound from tree metric

of the different bounds in comparison to their runtime. The newly proposed bounds, as well as *SLF*, are very fast, with varying degrees of accuracy. Although *CLB* is much faster than *BranchLB*, its accuracy is in many cases on par or only slightly worse. Note that a timeout of 120 seconds per graph pair was used for the computation of the exact graph edit distance for this experiment. For this reason, values for *BSS_GED* are not present for the datasets with larger graphs.

Q2: Evaluation of Runtime and Selectivity The runtime of the algorithms consists of three parts: (1) preprocessing and indexing, (2) filtering, and (3) verification. Preprocessing and indexing is performed only once, and this cost amortizes over many queries, while the time required to determine the candidate set and its size are crucial. The verification step requires to compute the exact graph edit distance and is usually most expensive and essentially depends on the number of candidates.

In the following, we investigate how well *EmbAssi* performs on range queries, how much of a speed-up can be achieved for existing pipelines when filtering with *EmbAssi* first, and compare to state-of-the-art approaches. We omit bounds that were shown in the previous experiments to have a poor accuracy or a very high runtime. Figure 6 shows the runtime for preprocessing, filtering and the average number of candidates per query for range queries with thresholds 1 to 5. The solid lines show the results, when using EmbAssi with *CLB* as a first filter, while the dotted line represent the original approaches. The solid red line shows the results using only *EmbAssi* with *CLB* and no further filters. *GSim* and *MLIndex* are shown with dashed lines, since they are stand-alone approaches. These two methods skip database graphs that are smaller than the given threshold. To obtain a valid candidate set, these graphs were added back after filtering. For *GSim* and *MLIndex* the preprocessing time is rather high and highly dependent on the maximum threshold for range search, which must be chosen in advance.

It becomes evident that *EmbAssi* significantly accelerates all methods across the various datasets. The preprocessing and filtering time of *EmbAssi* is very low: While filtering only takes a few milliseconds, preprocessing ranges from 0.01 to 2 seconds over the various datasets. *CStar* and *Branch* have the best selectivity, but they also employ both upper and lower bounds and need more time for filtering. The usage of *EmbAssi* heavily accelerates both methods, while even increasing the selectivity of

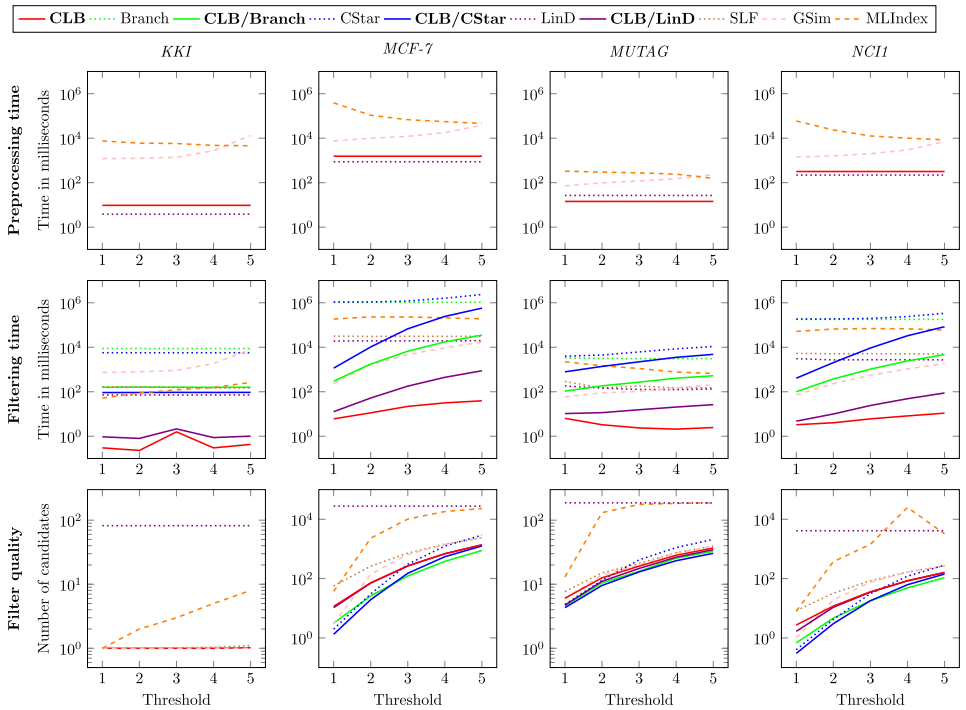


Fig. 6 Runtime and selectivity comparison of different filters. Preprocessing time, filtering time for 50 range queries (excluding verification), and the average number of candidates, that need to be verified, is shown. For the methods, that can be enhanced using *EmbAssi*, the solid line shows the advantage of pre-filtering with *CLB*, while the dotted line is the original approach

$CStar$ (as seen in Fig. 5, $CStarLB$ seems to be looser than *CLB* in general). Note, that *LinD* is an upper bound, so the candidate set consist of all graphs, that could not be reported as a result. In combination with *EmbAssi* it is only slightly worse than the other approaches regarding filter selectivity, while being very fast.

Considering the properties of the datasets and the performance, we observe that a larger set of vertex labels and a high variance among the vertex degrees seem to lead to a better filter quality. The larger the graphs, the greater the improvement in runtime during the filtering step.

Since competing approaches do not use the fast verification algorithm *BSS_GED* (Chen et al. 2019) a comparison of verification time would not be fair. On the various datasets the time for verification (of 50 queries with threshold 5) using the candidates of *CLB* ranged from around 35ms (KKI) to a maximum of 5s (MCF-7).

Combining these results, we can conclude that *EmbAssi* is well suited as pre-filtering for more effective computational demanding bounds. *EmbAssi* substantially reduces the filtering time and promises scalability even to very large datasets. We investigate this below.

Q3: Similarity Search on Very Large Datasets We investigate how well *EmbAssi* performs on very large graph databases using the datasets *Protein Com* and *ChEMBL*. Figure 7 shows the average number of candidates per query as reported by the different methods, as well as the time needed for preprocessing and filtering. *MLIndex* did not

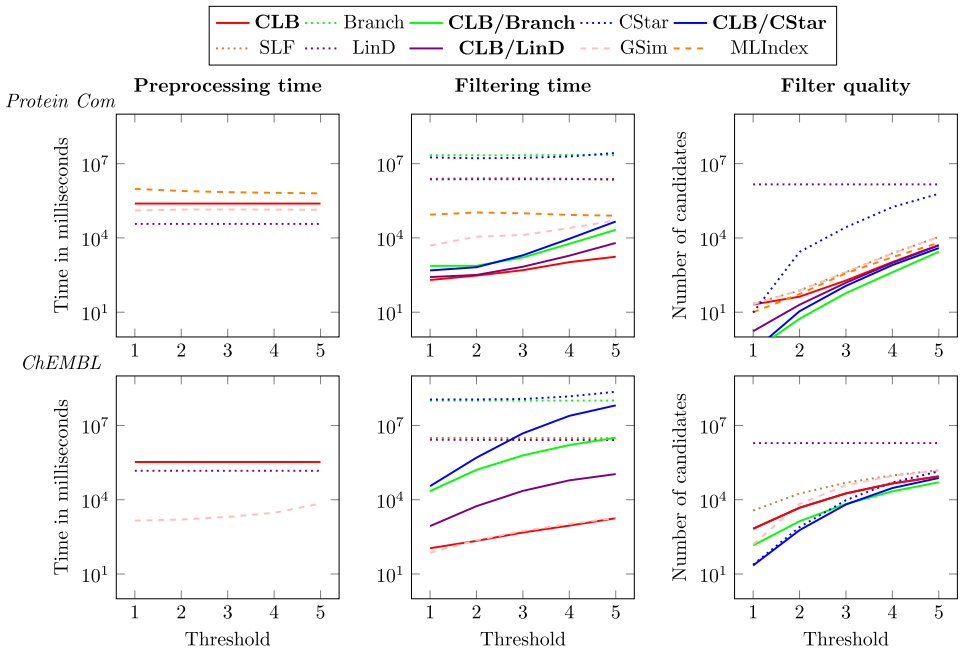


Fig. 7 Comparison of several different filters regarding their selectivity and runtime on datasets *Protein Com* and *ChEMBL*. The solid lines show the advantage of using *EmbAssi* with *CLB* as a pre-filter, while the dotted lines show the original approaches

Table 4 Runtime and number of candidates in k-nearest-neighbor search using *EmbAssi* and *BranchLB*

k	EmbAssi		BranchLB		|NN|
	|Cand|	Time (sec)	|Cand|	Time (sec)	
PTC_FM					
1	14.40	0.24	9.00	1.42	1.20
2	24.40	0.42	19.60	1.34	3.40
3	28.40	0.43	22.20	1.16	4.40
4	31.80	1.06	25.20	1.40	5.60
5	39.00	1.19	31.20	1.24	6.60
MUTAG					
1	6.80	0.15	6.80	0.87	1.60
2	9.80	0.22	9.80	0.99	3.80
3	14.00	0.25	14.00	1.31	5.80
4	14.00	0.49	14.00	1.03	5.80
5	15.80	0.55	15.80	1.08	7.40
MCF-7					
1	659.67	48.04	434.33	191.95	1.33
2	1114.33	152.43	737.00	244.42	2.33
3	1610.00	214.64	1045.67	410.11	4.00
4	1968.33	391.37	1380.33	577.75	10.00
5	2401.00	642.29	1696.33	678.06	10.67

finish on *ChEMBL* within a time limit of 24 hours (for threshold 1). For dataset *Protein Com* our new approach is not only much faster, but also provides a better filter quality than state-of-the-art methods. It can clearly be seen, that *EmbAssi* with *CLB* provides a substantial boost in runtime, while also improving the filter quality.

Q4: k-Nearest-Neighbor Search An advantage of *EmbAssi* is that it can also answer k-nn queries efficiently due to the use of the multi-step k-nearest neighbor search algorithm as described in Sect. 3.3. Table 4 compares the average number of candidates generated using *EmbAssi* (with *CLB*) and *BranchLB*, as well as the average time needed for answering a k-nn query. In both methods, candidate sets were verified using the faster exact graph edit distance computation *BSS_GED*. The last column shows the average number of nearest neighbors reported, which may be larger than k because of ties.

It can be seen, that *EmbAssi* provides a runtime advantage in k-nearest neighbor search, and the number of candidates generated is not much higher than when using *BranchLB*. For larger datasets, we expect the advantage of *EmbAssi* to be more significant. Further optimization of the approach is possible. For example, it might be beneficial to combine both methods and use *EmbAssi* in combination with tighter lower bounds such as *BranchLB* to reduce the number of exact graph edit distance computations.

7 Conclusions

We have proposed new lower bounds on the graph edit distance, which are efficiently computed, readily combined with indices, and fairly selective in filtering. This makes them ideally suitable as a pre-filtering step in existing filter-verification pipelines that do not scale to large databases. Our approach supports efficient k-nearest neighbor search using the optimal multi-step k-nearest neighbor search algorithm unlike many comparable methods. Other methods have to first perform a range query with a sufficient range and find the k-nearest neighbors among those candidates.

An interesting direction of future work is the combination and development of indices for computational demanding lower bounds such as those obtained from general assignment problems or linear programming relaxations. Efficient methods for similarity search regarding the Wasserstein distance have only recently been investigated (Backurs et al. 2020). Moreover, approximate filter techniques for the graph edit distance based on embeddings learned by graph neural networks were only recently proposed (Qin et al. 2020). With the increasing amount of structured data, scalability is a key issue in graph similarity search.

Acknowledgements This work was supported by the Vienna Science and Technology Fund (WWTF) through project VRG19-009. Additional funding was provided by the German Research Foundation (DFG) within the Collaborative Research Center SFB 876 *Providing Information by Resource-Constrained Data Analysis*, DFG project number 124020371, SFB projects A2 and A6, http://sfb876.tu-dortmund.de.

Funding Open access funding provided by University of Vienna.

References

Backurs A, Dong Y, Indyk P, Razenshteyn I, Wagner T (2020) Scalable nearest neighbor search for optimal transport. In: Int. Conf. Machine Learning, ICML, **119**, 497–506

Bai Y, Ding H, Bian S, Chen T, Sun Y, Wang W (2019) SimGNN: A neural network approach to fast graph similarity computation. In: ACM International Conference on Web Search and Data Mining, WSDM. https://doi.org/10.1145/3289600.3290967

Bause F, Blumenthal DB, Schubert E, Kriege NM (2021) Metric indexing for graph similarity search. In: SISAP 2021. Lecture Notes in Computer Science, vol. 13058 https://doi.org/10.1007/978-3-030-89657-7_24

Beygelzimer A, Kakade SM, Langford J (2006) Cover trees for nearest neighbor. In: Int. Conf. Machine Learning, ICML, vol. 148. https://doi.org/10.1145/1143844.1143857

Blumenthal D, Boria N, Gamper J, Bougleux S, Brun L (2019) Comparing heuristics for graph edit distance computation. VLDB J 29(1):419–458. https://doi.org/10.1007/s00778-019-00544-1

Bock HH (1974) Automatische Klassifikation. Vandenhoeck & Ruprecht, ???

Burkard RE, Dell'Amico M, Martello S (2012) Assignment Problems. SIAM, ???. https://doi.org/10.1137/1.9781611972238

Chang L, Feng X, Lin X, Qin L, Zhang W, Ouyang D (2020) Speeding up GED verification for graph similarity search. In: Int. Conf. Data Engineering, ICDE, pp. 793–804. https://doi.org/10.1109/ICDE48307.2020.00074

Chen X, Huo H, Huan J, Vitter JS (2019) An efficient algorithm for graph edit distance computation. Knowl-Based Syst 163:762–775. https://doi.org/10.1016/j.knosys.2018.10.002

Duan R, Su H-H (2012) A scaling algorithm for maximum weight matching in bipartite graphs. In: Symposium on Discrete Algorithms, SODA https://doi.org/10.1137/1.9781611973099.111

Ester M, Kriegel H, Sander J, Xu X (1996) A density-based algorithm for discovering clusters in large spatial databases with noise. In: Int. Conf. Knowledge Discovery and Data Mining (KDD), pp. 226–231

Garcia-Hernandez C, Fernández A, Serratosa F (2019) Ligand-based virtual screening using graph edit distance as molecular similarity measure. J Chem Inf Model 59(4):1410–1421. https://doi.org/10.1021/acs.jcim.8b00820

Gaulton A, Hersey A, Nowotka M, Bento AP, Chambers J, Mendez D, Mutowo P, Atkinson F, Bellis LJ, Cibrián-Uhalte E, Davies M, Dedman N, Karlsson A, Magariños MP, Overington JP, Papadatos G, Smit I, Leach AR (2016) The ChEMBL database in 2017. Nucleic Acids Res 45(D1):945–954. https://doi.org/10.1093/nar/gkw1074

Gouda K, Hassaan M (2016) CSI_GED: An efficient approach for graph edit similarity computation. In: Int. Conf. Data Engineering, ICDE. https://doi.org/10.1109/ICDE.2016.7498246

Kim J, Choi D, Li C: Inves: Incremental partitioning-based verification for graph similarity search. In: EDBT, pp. 229–240 (2019). https://doi.org/10.5441/002/edbt.2019.21

Kriege NM, Fey M, Fisseler D, Mutzel P, Weichert F (2018) Recognizing cuneiform signs using graph based methods. In: Int. Workshop on Cost-Sensitive Learning, COST@SDM. PMLR, **88**

Kriege NM, Giscard P, Bause F, Wilson RC: Computing optimal assignments in linear time for approximate graph matching. In: ICDM, pp. 349–358 (2019). https://doi.org/10.1109/ICDM.2019.00045

Kriege NM, Giscard P, Wilson RC. (2016) On valid optimal assignment kernels and applications to graph classification. In: Advances in Neural Information Processing Systems, pp. 1615–1623

Kriege NM, Johansson FD, Morris C (2020) A survey on graph kernels. Appl. Netw. Sci. 5(1):6. https://doi.org/10.1007/s41109-019-0195-3

Lerouge J, Abu-Aisheh Z, Raveaux R, Héroux P, Adam S (2017) New binary linear programming formulation to compute the graph edit distance. Pattern Recognit 72:254–265. https://doi.org/10.1016/j.patcog.2017.07.029

Le T, Yamada M, Fukumizu K, Cuturi M (2019) Tree-sliced variants of Wasserstein distances. In: Neural Information Processing Systems

Liang Y, Zhao P (2017) Similarity search in graph databases: A multi-layered indexing approach. In: Int. Conf. Data Engineering, ICDE. https://doi.org/10.1109/ICDE.2017.129

Li Y, Gu C, Dullien T, Vinyals O, Kohli P (2019) Graph matching networks for learning the similarity of graph structured objects. In: ICML

Morris C, Kriege NM, Bause F, Kersting K, Mutzel P, Neumann, M (2020) TUDataset: A collection of benchmark datasets for learning with graphs. In: ICML Workshop on Graph Representation Learning and Beyond, GRL+

Munkres JR (1957) Algorithms for the assignment and transportation problems. J Soc Ind Appl Math 5(1):32–38

Nasr R, Hirschberg DS, Baldi P (2010) Hashing algorithms and data structures for rapid searches of fingerprint vectors. J Chem Inf Model 50(8):1358–1368. https://doi.org/10.1021/ci100132g

Neuhaus M, Riesen K, Bunke H (2006) Fast suboptimal algorithms for the computation of graph edit distance. In: Structural, Syntactic, and Statistical Pattern Recognition, pp. 163–172. https://doi.org/10.1007/11815921_17

Qin Z, Bai Y, Sun Y (2020) GHashing: Semantic graph hashing for approximate similarity search in graph databases. In: ACM SIGKDD, pp. 2062–2072

Riesen K, Bunke H (2009) Approximate graph edit distance computation by means of bipartite graph matching. Image Vision Comput 27(7):950–959. https://doi.org/10.1016/j.imavis.2008.04.004

Riesen K, Ferrer M, Fischer A, Bunke H: Approximation of graph edit distance in quadratic time. In: Graph-Based Representations in Pattern Recognition, pp. 3–12 (2015)

Schubert E, Zimek A, Kriegel H (2014) Local outlier detection reconsidered: a generalized view on locality with applications to spatial, video, and network outlier detection. Data Min Knowl Discov 28(1):190–237. https://doi.org/10.1007/s10618-012-0300-z

Schubert E, Zimek A (2019) ELKI: A large open-source library for data analysis - ELKI release 0.7.5 Heidelberg. CoRR arXiv: abs/1902.03616

Seidl T, Kriegel H (1998) Optimal multi-step k-nearest neighbor search. In: SIGMOD Int. Conf. Management of Data, pp. 154–165. https://doi.org/10.1145/276304.276319

Seidl M, Wieser E, Zeppelzauer M, Pinz A, Breiteneder C (2015) Graph-based shape similarity of petroglyphs. In: ECCV Workshops Computer Vision, pp. 133–148

Semple C, Steel M (2003) Phylogenetics. Oxford lecture series in mathematics and its applications. Oxford University Press, ???

Sibson R (1973) SLINK: An optimally efficient algorithm for the single-link cluster method. The Computer Journal 16(1):30–34. https://doi.org/10.1093/comjnl/16.1.30

Stöcker BK, Schäfer T, Mutzel P, Köster J, Kriege, NM, Rahmann S (2019) Protein complex similarity based on Weisfeiler-Lehman labeling. In: 12th Int. Conf. Similarity Search and Applications, SISAP, **11807**, 308–322. https://doi.org/10.1007/978-3-030-32047-8_27

Wang G, Wang B, Yang X, Yu G (2012) Efficiently indexing large sparse graphs for similarity search. IEEE Trans Knowl Data Eng 24(3):440–451. https://doi.org/10.1109/TKDE.2010.28

Wang X, Ding X, Tung A, Ying S, Jin H (2012) An efficient graph indexing method. In: Int. Conf. Data Engineering, ICDE https://doi.org/10.1109/ICDE.2012.28

Wu Z, Pan S, Chen F, Long G, Zhang C, Yu PS (2021) A comprehensive survey on graph neural networks. IEEE Trans Neural Networks Learn Syst 32(1):4–24. https://doi.org/10.1109/TNNLS.2020.2978386

Xiao B, Cheng J, Hancock ER (2013) Graph-based Methods in Computer Vision: Developments and Applications. Premier reference source. Information Science Reference, ???

Yang L, Zou L (2021) Noah: Neural-optimized A* search algorithm for graph edit distance computation. In: 2021 IEEE 37th International Conference on Data Engineering (ICDE), pp. 576–587. https://doi.org/10.1109/ICDE51399.2021.00056

Zeng Z, Tung AKH, Wang J, Feng J, Zhou L (2009) Comparing stars: On approximating graph edit distance. Proc. VLDB Endow. 2(1):25–36. https://doi.org/10.14778/1687627.1687631

Zhao X, Xiao C, Lin X, Liu Q, Zhang W (2013) A partition-based approach to structure similarity search. Proc VLDB Endow 7(3):169–180. https://doi.org/10.14778/2732232.2732236

Zhao X, Xiao C, Lin X, Wang W (2012) Efficient graph similarity joins with edit distance constraints. In: Int. Conf. Data Engineering, ICDE https://doi.org/10.1109/ICDE.2012.91

Zheng W, Zou L, Lian X, Wang D, Zhao D (2015) Efficient graph similarity search over large graph databases. IEEE Trans Knowl Data Eng 27(4):964–978. https://doi.org/10.1109/TKDE.2014.2349924

A.3 Approximating the Graph Edit Distance with Compact Neighborhood Representations

Authors Franka Bause*, Christian Permann*, and Nils M. Kriege

Publication Venue Published in *Machine Learning and Knowledge Discovery in Databases. Research Track*, LNCS 14945, pages 300–318 (2024) by Springer Nature.

Acknowledgment Reproduced with permission from Springer Nature.

DOI/URL https://doi.org/10.1007/978-3-031-70362-1_18

Extended Version https://arxiv.org/abs/2312.04123

Authors Contributions

Franka Bause* developed the idea of neighborhood trees together with CP, drafted, wrote, and revised the manuscript with input from the other authors, implemented the comparison methods, and conducted the experimental evaluation.

Christian Permann* developed the idea of neighborhood trees together with FB, implemented large parts of the framework, optimized the implementation, and helped in drafting and writing the manuscript.

Nils M. Kriege supervised the project, gave valuable feedback, helped to develop the proof of Theorem 2, and to revise the manuscript.

Reference in Thesis [BPK24]

Abstract The graph edit distance, used for comparing graphs in various domains, is often approximated due to its high computational complexity. Widely used heuristics search for an optimal assignment of vertices based on the distance between local substructures. However, some sacrifice accuracy by only considering direct neighbors, while others demand intensive distance calculations. Our method abstracts local sub- structures to neighborhood trees, efficiently comparing them using tree matching techniques. This yields a ground distance for vertex mapping, delivering high quality approximations of the graph edit distance. By limiting the maximum tree height, our method offers to balance accuracy and computation speed. We analyze the running time of the tree matching method and propose techniques to accelerate

*Equal contribution.

computation in practice, including compressed tree representations, tree canonization to identify redundancies, and caching. Experimental results demonstrate significant improvements in the trade-off between running time and approximation quality compared to existing state-of-the-art approaches.

Approximating the Graph Edit Distance with Compact Neighborhood Representations

Franka Bause[1,2(✉)], Christian Permann[3,4], and Nils M. Kriege[1,5]

[1] Faculty of Computer Science, University of Vienna, Vienna, Austria
{franka.bause,nils.kriege}@univie.ac.at
[2] UniVie Doctoral School Computer Science, University of Vienna, Vienna, Austria
[3] Department of Pharmaceutical Sciences, University of Vienna, Vienna, Austria
christian.permann@univie.ac.at
[4] Research Platform NeGeMac, University of Vienna, Vienna, Austria
[5] Research Network Data Science, University of Vienna, Vienna, Austria

Abstract. The graph edit distance, used for comparing graphs in various domains, is often approximated due to its high computational complexity. Widely used heuristics search for an optimal assignment of vertices based on the distance between local substructures. However, some sacrifice accuracy by only considering direct neighbors, while others demand intensive distance calculations. Our method abstracts local substructures to neighborhood trees, efficiently comparing them using tree matching techniques. This yields a ground distance for vertex mapping, delivering high quality approximations of the graph edit distance. By limiting the maximum tree height, our method offers to balance accuracy and computation speed. We analyze the running time of the tree matching method and propose techniques to accelerate computation in practice, including compressed tree representations, tree canonization to identify redundancies, and caching. Experimental results demonstrate significant improvements in the trade-off between running time and approximation quality compared to existing state-of-the-art approaches.

Keywords: Graph edit distance · Bipartite graph matching · Trees

1 Introduction

Graphs have become prevalent in modeling structured data, with applications in web content mining [29], pattern recognition [10], molecular property prediction [11], and various other domains. Many tasks and applications require quantifying the similarity of two graphs. One of the most prominent measures of graph similarity is the *graph edit distance* (GED), which measures the cost of transforming one graph into another through predefined edit operations. However, exact computation of the GED is limited due to its NP-hard [37] nature. To address

F. Bause and C. Permann—Contributed equally to this work.

A. Bifet et al. (Eds.): ECML PKDD 2024, LNAI 14945, pp. 300–318, 2024.
https://doi.org/10.1007/978-3-031-70362-1_18

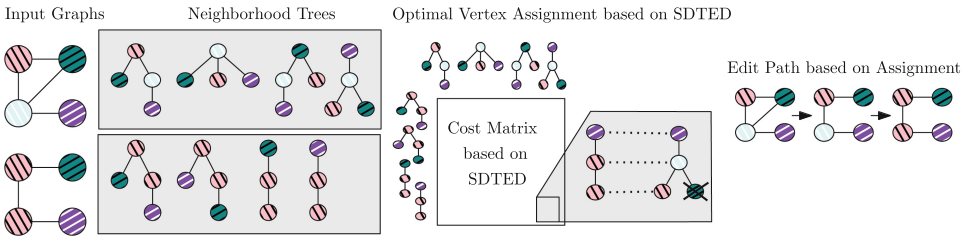


Fig. 1. Overview of our approach. To approximate the GED between two graphs, an optimal assignment between their vertices is computed based on a cost matrix. The entries of the matrix correspond to the SDTEDs between the neighborhood trees of the associated vertices. An edit path is derived from the assignment, and the cost of this (suboptimal) edit path is the approximated GED.

this, fast heuristic algorithms such as *bipartite graph matching* (BGM) [28] have been developed, which approximates the GED by deriving a suboptimal edit path from an optimal (linear) assignment between the vertices of the graphs. The method provides a fast approximation widely used in many domains [34]. However, some applications require a higher approximation quality, such as graph clustering, k-nn classification or similarity search in graph databases. Extensions of BGM capture only limited structural information using walks [12] or require exact GED computation between subgraphs [8] leading to a drastic increase in both memory and time complexity. To improve the trade-off between computation time and approximation quality, we propose a new GED heuristic based on BGM, integrating local information in the form of *neighborhood trees*. We compare neighborhood trees by a *structure and depth preserving tree edit distance* (SDTED), which is highly efficient compared to computing the exact GED, while retaining the benefits of incorporating expressive structural information.

Neighborhood trees are inspired by *unfolding trees*, which are related to the Weisfeiler-Leman algorithm. The technique has recently become popular in graph learning, where it forms the basis of successful graph kernels [19,33] and is fundamental to the expressivity analysis of graph neural networks [23,24,36]. The discrete colors representing unfolding trees have been used to obtain a highly efficient heuristic for the GED ignoring their subtle structural differences at the expense of approximation quality [17]. While unfolding trees are a suitable concept for understanding the structure encoded by iterative methods based on direct neighborhoods, they are rarely generated explicitly, since they quickly grow in size with increasing height. Our concept of neighborhood trees overcomes this problem by pruning repeated vertices and allowing compact representations without redundancy amenable to efficient tree matching.

Our Contribution. We make the following contributions.

1. We use tree structures encoding node neighborhoods and their SDTED in the BGM framework to approximate the GED (see Fig. 1 for an overview).
2. We propose *neighborhood trees*, a compact variation of unfolding trees.

3. We thoroughly analyze the running time of the SDTED and improve it to $O(n^2\Delta)$ for two trees with n vertices and maximum degree Δ, reducing it by a factor of $\Delta^2 h$ over the best known result [30], where h is the tree height.
4. We employ caching and compression to significantly accelerate computation.
5. We show that our approach outperforms state-of-the-art GED approximation methods regarding approximation quality, while being computed efficiently.

2 Preliminaries

Graph Theory. A *graph* $G = (V, E, \mu, \nu)$ consists of a set of vertices V, a set of edges $E \subseteq V \times V$ between them, and labeling functions $\mu\colon V \to L$ and $\nu\colon E \to L$ for the vertices and edges, respectively. Here, L is a set of categorical labels. We consider undirected graphs, denoting an edge between u and v by $uv = vu$. Vertices and edges of a graph G are denoted by $V(G)$ and $E(G)$, respectively. The *neighbors* of a vertex $u \in V$ are denoted by $N(u) = \{v \mid uv \in E\}$. A *path* is a sequence of vertices $(v_0, \ldots, v_n)$ such that $v_i v_{i+1} \in E$ for $0 \leq i < n$, with vertices and edges connecting consecutive vertices being *contained* in the path. The *length* of a path is the number of edges it contains. A *shortest path* between two vertices $u, v \in V$ is a path $(u, \ldots, v)$ of minimum length. The *diameter* of a graph G is the maximum shortest-path length between any two vertices in G, denoted by $\mathrm{diam}(G)$. A *rooted tree* T is a connected, acyclic graph with a distinct vertex $r \in V(T)$ referred to as *root*, denoted by $r(T)$. For $v \in V(T)\backslash\{r\}$ the *parent* $p(v)$ is the unique vertex $u \in N(v)$ closer to r than v, and $\forall v \in V(T)$ the *children* are defined as $\mathrm{chi}(v) = N(v)\backslash\{p(v)\}$. A node v is a *leaf* if $\mathrm{chi}(v) = \emptyset$. The leaves of a tree T are denoted by $L(T) = \{v \in V(T) \mid \mathrm{chi}(v) = \emptyset\}$.

Two graphs G and H are *isomorphic*, denoted by $G \cong H$, if there exists a bijection $\Phi\colon V(G) \to V(H)$ such that (i) $\forall u, v \in V(G)\colon \mu(v) = \mu(\Phi(v))$, (ii) $\Phi(u)\Phi(v) \in E(H) \Leftrightarrow uv \in E(G)$, and (iii) $\forall uv \in E(G)\colon \nu(uv) = \nu(\Phi(u)\Phi(v))$. For two rooted trees T and T', also $\Phi(r(T)) = r(T')$ must hold.

Graph Edit Distance. The graph edit distance (GED) measures the cost of transforming one graph into the other through deletion, insertion, and relabeling of vertices and edges. An *edit path* between G and H is a sequence $(e_1, e_2, \ldots, e_k)$ of edit operations that, when applied sequentially to G, yield a graph $G' \cong H$. Edit operations have non-negative costs defined by the edit cost function c. The GED is the cost of a cheapest edit path. Formally, the *graph edit distance* between two graphs G and H is defined as $\mathrm{GED}(G, H) = \min\left\{\sum_{i=1}^{k} c(e_i) \;\middle|\; (e_1, ..., e_k) \in \Upsilon(G, H)\right\}$, where $\Upsilon(G, H)$ denotes the set of all possible edit paths from G to H. As computing the GED is NP-hard [37], it is often approximated. Many heuristics rely on finding an optimal assignment between the vertices of the graphs. Let A and B be two sets with $|A| = |B| = n$ and $c\colon A \times B \to \mathbb{R}$ a ground cost function. An *assignment* between A and B is a bijection $\pi\colon A \to B$. The *cost* of an assignment π is $c(\pi) = \sum_{a \in A} c(a, \pi(a))$. The *assignment problem* is to find an assignment with minimum cost and can be solved in cubic time using the Hungarian method [16,25].

Bipartite Graph Matching. An upper bound for the GED between two graphs can be obtained from any edit path between them, which can be derived from an optimal assignment of their vertices. Riesen and Bunke [28] constructed a $k \times k$ cost matrix $\boldsymbol{C}$, with $k = n + m$ for two graphs with n and m vertices, respectively. For simplicity, we present a reduced matrix with $k = \max\{n, m\}$ that is sufficient for metric edit costs [32]. Due to the symmetry of the GED no vertices need to be inserted. Assume w.l.o.g. that $n \geq m$, then we obtain a quadratic $n \times n$ cost matrix by adding $n - m$ columns representing deletion of vertices in G by defining the cost matrix $\boldsymbol{C}$ (Eq. 1). The entries of $\boldsymbol{C}$ can be interpreted as follows: Values $c_{i,j}$ (left part) represent costs for replacing vertex i of G with vertex j of H. The entries $c_{i,\epsilon}$ (right part) are costs for deleting vertex i of G and its incident edges. Based on $\boldsymbol{C}$ an optimal assignment is computed. An equivalent solution can be obtained from an $n \times m$ cost matrix by a more involved reduction applicable even for non-metric edit costs [7].

$$\boldsymbol{C} = \left[\begin{array}{ccc|ccc} c_{1,1} & c_{1,2} & \cdots & c_{1,m} & c_{1,\epsilon} & \cdots & c_{1,\epsilon} \\ c_{2,1} & c_{2,2} & \cdots & c_{2,m} & c_{2,\epsilon} & \cdots & \vdots \\ \vdots & \vdots & \ddots & \vdots & \vdots & \ddots & \vdots \\ c_{n,1} & c_{n,2} & \cdots & c_{n,m} & c_{n,\epsilon} & \cdots & c_{n,\epsilon} \end{array}\right] \tag{1}$$

An edit path can be derived from a vertex assignment $\pi\colon V(G) \to V(H)$ as follows [28]: Any vertex $v \in V(G)$ with $\pi(v) \notin V(H)$ is deleted, $v \in V(H)$ with $\pi^{-1}(v) \notin V(G)$ is inserted, $v \in V(G)$ with $\pi(v) \in V(H)$ and $\mu(v) \neq \mu(\pi(v))$ is relabeled. Any edge $uv \in E(G)$ with $\pi(u)\pi(v) \notin E(H)$ is deleted, $uv \in E(H)$ with $\pi^{-1}(u)\pi^{-1}(v) \notin E(G)$ is inserted, $uv \in E(G)$ with $\pi(u)\pi(v) \in E(H)$ and $\nu(uv) \neq \nu(\pi(u)\pi(v))$ is relabeled. The sum of edit costs is the approximated GED. Since the edit path might not be optimal, these costs are an upper bound.

Weisfeiler-Leman Unfolding Trees. The Weisfeiler-Leman *unfolding trees* encode neighborhoods of increasing radius by a tree structure. Let $\phi\colon V(T) \to V(G)$ be a mapping such that $\phi(n) = v$ if the node n in $V(T)$ represents the vertex v in $V(G)$. Mathematically, the *unfolding tree* F_i^v with height i of the vertex $v \in V(G)$ is defined recursively as the tree with a root n_v with $\phi(n_v) = v$ and child subtrees F_{i-1}^w for all $w \in N(v)$, and $F_0^v = (\{n_v\}, \emptyset)$. The labels of the original graph are preserved.

Structure and Depth Preserving Tree Edit Distance. The *structure and depth preserving tree edit distance* (SDTED) [30] computes the distance between two labeled, rooted trees, preserving parent-child-relations, as well as the depth of vertices. It is defined as $\text{SDTED}(T, T') := \min\{c'(M) \mid M \in \text{SDM}(T, T')\}$, where $\text{SDM}(T, T')$ denotes the set of all structure and depth preserving mappings between T and T'. A mapping $M \subseteq V(T) \times V(T')$ is *structure and depth preserving*, iff (i) $\forall (u, u'), (v, v') \in M\colon\ u = v \Leftrightarrow u' = v'$, (ii) $(r(T), r(T')) \in M$, and (iii) $\forall (u, u') \in M\colon\ (p(u), p(u')) \in M$. Let $M_1 = V(T) \backslash \{m_1 \mid (m_1, m_2) \in M\}$ and $M_2 = V(T') \setminus \{m_2 \mid (m_1, m_2) \in M\}$ be the vertices not in the mapping.

The cost c' of the mapping M is defined as

$$c'(M) = \underbrace{\sum_{(u,v)\in M} c(\mu(u),\mu(v))}_{\text{Relabeling}} + \underbrace{\sum_{u\in M_1} c(u,\epsilon)}_{\text{Deletion}} + \underbrace{\sum_{v\in M_2} c(\epsilon,v)}_{\text{Insertion}}.$$

We can respect the label of an edge $p(v)v$ by associating the cost for its relabeling with the endpoint v. Given two trees, the SDTED can be computed in a bottom-up fashion from optimal assignments between the children of vertices at the same level until reaching the roots. For our implementation we included edge costs and store the costs of all computed pairs of trees and subtrees using a lexicographic encoding, see Sect. 4.6.

3 Related Work

Exact Methods for Computing the GED. Computing the GED is a combinatorial optimization problem. Different standard approaches exist and have been adapted specifically for the GED. Some exact methods rely on backtracking search, e.g., A*-GED, DF-GED [1], CSI-GED [14] or BSS_GED [9], while others employ integer linear programming formulations, like BIP-GED [20] or MIP-GED [6]. Despite their accuracy, exact approaches become infeasible for large graphs due to their complexity, and approximate methods are widely employed [34].

Approximate Methods for the GED. Many approximation methods [4,5,8,28,37] are based on optimal assignments, computing an assignment between the vertices of the graphs to find an upper or lower bound for the GED. Typically, faster methods are suitable only for graphs with discrete labels and uniform edit costs [4]. Riesen and Bunke [28] introduced the foundational concept of bipartite graph matching (BGM), estimating the costs for the assignment problem based on the vertices assigned to each other and their incident edges. This general approach can handle continuous labels and has been further refined to yield a tight lower bound for arbitrary (metric) edit costs, known as Branch [5].

While restricting to vertices and their incident edges allows obtaining lower bounds for the GED, better approximations can be achieved using larger substructures. Carletti et al. [8] proposed to use neighborhood subgraphs, which are defined for a vertex v of G as the subgraph induced in G by the vertices with a shortest-path distance from v of at most h, where h is a radius parameter. The cost for assigning a node v to a node v' is obtained by comparing their neighborhood subgraphs using the exact GED with the additional constraint that v is assigned to v'. However, the exact GED computation leads to a running time exponential in the size of the subgraphs, drastically increasing the computational complexity of the method. Gaüzère et al. [12] associate bags of walks B_v with each vertex v, containing all label sequences $(\mu(v_0), \nu(v_0v_1), \mu(v_1), \ldots, \mu(v_h))$ associated with walks $(v_0, v_1, \ldots, v_h)$ of length h starting at the vertex $v = v_0$.

The cost for assigning v to v' is obtained by comparing B_v and $B_{v'}$. To avoid enumerating an exponential number of walks, the assignment cost is estimated from the hth powers of the adjacency matrices $\boldsymbol{A}_G$, $\boldsymbol{A}_H$ and $\boldsymbol{A}_{G\times H}$ of G, H and their direct product graph $G \times H$, respectively. This approach loses local structure information, as only the labels of the start and end vertices are considered, omitting intermediate vertices. Moreover, the approximation quality decreases for walks with five or more vertices due to the phenomenon of *tottering* [12]. In Table 1, we compare the theoretical complexity of the methods based on BGM, closely related to our approach.

GED approximation methods not based on BGM are often multiple orders of magnitude slower, with only marginal improvements in accuracy [4]. These methods, such as the LP relaxations of ILP formulations [20], may yield tighter lower bounds at drastically increased runtime.

Weisfeiler-Leman Based Methods for GED Approximation and Graph Similarity. Our concept of neighborhood trees is inspired by Weisfeiler-Leman unfolding trees. Graph and vertex similarities based on the Weisfeiler-Leman algorithm have been studied extensively in machine learning [21]. Several graph kernels count matching Weisfeiler-Leman colors [19], use them to compute optimal vertex assignments [18] or define similarities between colors, e.g., by matching unfolding trees [30]. Kriege et al. [17] proposed a GED approximation based on finding an optimal vertex assignment regarding a tree metric generated by the Weisfeiler-Leman algorithm. This approach employs matching Weisfeiler-Leman colors and implicitly compares the unfolding trees of two vertices level-wise (from root to leaves), determining similarity as the number of levels up to which the unfolding trees are isomorphic. This results in a coarse similarity, where vertices with different labels, but equal neighborhoods are considered less similar than vertices with the same label and completely different neighborhoods. Although faster than comparable BGM methods, it is less accurate on some datasets.

Graph neural networks have been used to predict the GED for pairs of graphs [3]. However, these methods do not yield an edit path and cannot guarantee that the result is a lower or upper bound for the GED. A recent method attempting to learn an edit path [26] suffers from high running time, and from the fact that datasets annotated with ground-truth GED are rare.

Discussion. Methods for approximating the GED inherently encounter a trade-off between approximation quality and running time. Among these, algorithms following the framework of BGM show particular promise in this regard [4]. However, in certain applications, the approximation quality of standard BGM algorithms is not sufficient. Current approaches using larger substructures [8,12] rely on computationally expensive cost functions. While this can potentially negate the advantage of BGM in terms of running time, the time complexity of these methods has often not been thoroughly analyzed. Therefore, we provide an analysis here. The method based on subgraphs [8] computes the exact GED between subgraphs of size Δ^h, where Δ is the maximum degree and h the radius parameter. As exact GED algorithms require exponential time for each of the

$|V|^2$ subgraph pairs, the running for computing the cost matrix time becomes $O(|V|^2 \exp(\Delta^h))$. For the method based on walks [12], the hth power of the adjacency matrices of both graphs and their product graph is computed. Using exponentiation by squaring $O(\log(h))$ matrix multiplications are needed, where the matrix size is $n^2 \times n^2$ for the product graph of two graphs with n vertices. From this, all entries of the cost matrix can be derived, leading to a total running time of $O(\log(h)|V|^{2\omega})$, with ω being the exponent of matrix multiplication. The summarized results are shown in Table 1, providing simplified running times for the case of input graphs with bounded-degree, which are relevant as the GED is often applied to compare graphs with bounded degree such as molecular graphs.

While our approach is less efficient in terms of time complexity compared to standard BGM, it proves to be more efficient or competitive with existing approaches using larger substructures. In Sect. 5 we demonstrate experimentally that our method outperforms these state-of-the-art approximation methods in terms of approximation quality while maintaining computational efficiency, offering a favorable trade-off.

Table 1. Time complexity of computing the cost matrix $\boldsymbol{C}$ for GED approximations based on BGM. Here, Δ denotes the maximum degree, h is the radius of the subgraphs/walk length and ω is the exponent of matrix multiplication (best algorithms achieve $\omega < 2.38$, but typically $\omega \approx 2.81$ in practice [4]). In addition, we consider the complexity, when Δ is bounded by a constant.

Method	General graphs	Bounded-degree graphs
BGM [28]	$O(\|V\|^2\Delta^3)$	$O(\|V\|^2)$
Walk [12]	$O(\log(h)\|V\|^{2\omega})$	$O(\log(h)\|V\|^{2\omega})$
Subgraph [8]	$O(\|V\|^2 \exp(\Delta^h))$	$O(\|V\|^2 \exp(h))$
Ours [Theorem 3]	$O(\|V\|^2\|E\|^2\Delta)$	$O(\|V\|^4)$

4 Neighborhood Trees for Graph Matching

Accurate approximation of the GED through BGM requires representations of sufficiently large and complex vertex neighborhoods, which allow the efficient computation of suitable and expressive distances. We propose to use trees for representing neighborhoods and use the SDTED between such trees as a cost function for the assignment. See Fig. 1 and Algorithm 1 for an overview of our approach. By leveraging the structured nature of rooted trees, we can efficiently compute distances, leading to significant speed-ups compared to neighborhood subgraphs. We explore traditional unfolding trees, which rapidly grow in size with increasing height and store a large amount of redundant information. To address this issue, we propose *k-redundant neighborhood trees*, which are equivalent to Weisfeiler-Leman unfolding trees for sufficiently large k while exhibiting less redundancy for smaller values of k, resulting in more compact representations. Of particular interest are 0-redundant neighborhood trees, which represent

Algorithm 1. GED approximation algorithm

```
function approximateGED(Graph G, Graph H, height h, k)
    assert |V(G)| ≥ |V(H)| by swapping G and H if necessary
    C ← empty |V(G)| × |V(G)| cost matrix
    for each u ∈ V(G) do T_u ← buildNT(G, u, h, k)              ▷ for each vertex
    for each v ∈ V(H) do T_v ← buildNT(H, v, h, k)              ▷ build k-NTs of height h
    for each u ∈ V(G) do
        for each v ∈ V(H) do
            C_{u,v} ← SDTED(T_u, T_v)                           ▷ fill cost matrix with SDTED values
    for each u ∈ V(G) do                                        ▷ deletion costs for vertices of larger graph
        C_{u,v} ← DeletionCosts(T_u) for v ∉ V(H)               ▷ (right part of matrix in Eq. 1)
    assignment ← solveAssignment(C)
    editpath ← deriveEditPath(assignment)
    return computeCost(editpath)
```

a vertex with its surroundings through a rooted tree of shortest paths up to a maximum length. By bounding the tree height, the trade-off between more accurate results and faster execution can effectively be controlled.

(a) G (b) Unfolding tree (c) 0-NT (d) cNT (e) 1-NT

Fig. 2. Graph G and neighborhood representations (height 2) of upper left vertex.

4.1 Definitions and Properties

A k-redundant neighborhood tree can be derived from the corresponding unfolding tree by removing nodes that occurred more than k levels earlier, along with their associated subtrees. Here, $\text{depth}(v)$ denotes the length of the path from vertex v to the root, and $\phi(v)$ denotes the original vertex in $V(G)$ represented by v in the unfolding or neighborhood tree.

Definition 1 (k-redundant Neighborhood Tree, k-NT). *For $k \geq 0$, the* k-redundant neighborhood tree *of a vertex $v \in V(G)$ with height i, denoted by $T^v_{i,k}$, is defined as the subtree of the unfolding tree F^v_i induced by the vertices $u \in V(F^v_i)$ satisfying $\forall w \in V(F^v_i)\colon \phi(u) = \phi(w) \Rightarrow \text{depth}(u) \leq \text{depth}(w) + k$.*

For $k \geq i$, the k-NT is equivalent to the Weisfeiler-Leman unfolding tree. We refer to the 0-NT simply as *neighborhood tree* (NT), see Fig. 2 for an example. In the neighborhood tree of a vertex v, the edges connecting two vertices with the same shortest-path distance to v are not be represented, e.g., the diagonal

edge between the blue and the green vertex of graph G in Fig. 2. Hence, we also investigate 1-redundant neighborhood trees (1-NTs), which include such edges. Note that of course, only edges in the same connected component as the vertex can be present in a neighborhood tree.

A k-NT may contain multiple instances of the same vertex. Two vertices u and v with $\phi(u) = \phi(v)$ and $\text{depth}(u) = \text{depth}(v)$ are duplicates of a vertex on the same level, and following the definition, the subtrees rooted at u and v are isomorphic. This means, the tree contains duplicate subtrees increasing its size without providing additional information. Therefore, we propose a *compressed neighborhood tree* (cNT) to avoid this redundancy. Given two vertices u and v on the same level of a rooted tree T, by *merging u onto v* we refer to the operation that adds the edge $p(u)v$ to T and deletes the subtree rooted at u. When merging two vertices, we assume that no vertices with lower depth can be merged.

Definition 2 (Compressed k-Redundant Neighborhood Tree, cNT). *The* compressed neighborhood tree $R^v_{i,k}$ *of $v \in V(G)$ is the reduction of $T^v_{i,k}$ obtained by merging vertices u onto vertices w whenever $\phi(u) = \phi(w)$ and* $\text{depth}(u) = \text{depth}(w)$.

The rooted subtrees of two vertices that are merged are isomorphic leading to a well-defined compressed neighborhood tree independent of the merging order of vertices at the same level. Size and height of a cNT are bounded as follows.

Theorem 1. *The cNT of a k-redundant neighborhood tree rooted at any vertex of a connected graph G has size at most $O(|E(G)| \cdot (k+1))$ and height at most* $\text{diam}(G) + k$, *where* $\text{diam}(G)$ *is the diameter of G.*

Proof. A cNT $R^v_{\infty,0}$ rooted at vertex $v \in V(G)$ includes all shortest paths from v to any vertex u with length at most $\text{diam}(G)$. As the parameter k in k-NTs allows for up to k repetitions of vertices in subsequent layers, at most k layers are added to the compressed k-NT through repetition of the vertex $w \in V(G)$ where $\forall u \in V(G)\colon \text{depth}(u) \leq \text{depth}(w)$. Thereby, the maximum height is bounded by $\text{diam}(G) + k$. Since no vertex can occur multiple times in a layer due to compression and a vertex can only occur in $1+k$ consecutive layers, we can infer an upper bound on the size of any compressed k-NT. Any vertex $u \in V(G)$ can occur at most $k+1$ times together with its incident edges. As $|E(G)| \geq |V(G)|+1$ since G is connected, the size of any $R^v_{i,k}$ is bounded by $O(|E(G)| \cdot (k+1))$.

Theorem 1 holds for connected graphs. If G consists of multiple connected components, each can be processed independently and the theorem applies for each component. Restricting the maximum height reduces the tree size further by pruning deeper levels.

4.2 Creating Compressed Neighborhood Trees

The compressed k-redundant neighborhood tree $R^v_{i,k}$ can be created directly, which is more efficient than deriving it from the full tree. Algorithm 2 shows how

Algorithm 2. Creation of compressed k-NT (cNT)

```
function BUILDNT(Graph G, vertex w, height h, k)
    T ← cTree(w)                                  ▷ initialize cNT with root w
    D ← empty dictionary                          ▷ initialize depth dictionary
    D(w) ← 0
    for i ← 1, ..., h do
        F ← empty dictionary                      ▷ initialize found dictionary
        for each v ∈ L(T) do
            for each u ∈ N(φ(v)) do
                if u not in D then                                  ▷ new vertex
                    D(u) ← i
                if D(u) + k ≥ i then
                    if u not in F then                              ▷ new on this depth
                        add new vertex c to V(T)
                        φ(c) ← u
                        F(u) ← c
                    add new edge vF(u) to E(T)
    return T
```

to generate compressed trees according to Definitions 1 and 2 with a maximum height h. The maximum height is naturally bounded by the diameter of G. Hence, we set the parameter h to $\min\{h, \mathrm{diam}(G) + k\}$ and assume $h \leq \mathrm{diam}(G) + k$ when the algorithm is applied. For the creation of a tree the chosen vertex is set as its root and then the neighborhood of this vertex is explored level-wise in a breadth-first search fashion. We build the cNT by keeping track of the first level at which a vertex v was found through $D(v)$, and whether a vertex was already found at the current level through $F(v)$. Then a corresponding vertex has already been created and only the missing edge is inserted. This continues until the maximum height h is reached. With these techniques Algorithm 2 constructs a compressed neighborhood tree of a given vertex in time $O(|E(G)| \cdot (k+1))$.

4.3 Neighborhood Tree Edit Distance

We compare (compressed) neighborhood trees using the SDTED to obtain a distance between vertex pairs. Vertices further away should have a smaller influence on the similarity of two vertices than the vertices themselves or their direct neighbors. Therefore, we introduce a sequence of weights $\boldsymbol{\lambda} = (\lambda_1, \lambda_2, \dots)$, where λ_i controls the contribution of the ith level of the neighborhood tree, multiplying the edit costs of the nodes at depth i by λ_i. We propose setting $\lambda_i = w^i$ for $0 \leq w \leq 1$, where w controls the influence of vertices with increasing depth. Choosing $w = 1$ yields the unweighted case, while $w < 1$ reduces the influence of vertices further away.

The SDTED can be computed recursively starting from the root vertices and solving optimal assignments between their children, where the cost for matching two children depends on the SDTED of their subtrees. Schulz et al. [30] used this

approach with memoization to obtain a runtime of $O(nn'h\Delta^3)$ for two unfolding trees with n and n' vertices, height h and a maximum degree of Δ. We improve this by using techniques for efficient tree matching and refining the analysis.

Theorem 2. *The SDTED of two trees with n and n' vertices and maximum degree Δ can be computed in $O(nn'\Delta)$ time.*

Proof. Consider two trees $T = (V, E)$, $T' = (V, E')$ with n and n' vertices, respectively. For each pair of vertices $(i, j) \in V \times V'$ with $\text{depth}(i) = \text{depth}(j)$, a linear assignment problem has to be solved. Let n_i and n_j denote the number of children of i and j, respectively. Introducing placeholder vertices for insertion and deletion leads to a squared cost matrix with $n_i + n_j$ rows and columns [30]. However, we can find an equivalent solution using the reduction of Bougleux et al. [7] resulting in an $n_i \times n_j$ cost matrix $\boldsymbol{C}$. The resulting problems can be solved by maximum weight matching algorithms for unbalanced bipartite graphs in time $O(ms + s^2 \log s)$, where m is the number of edges, i.e., non-zero entries in $\boldsymbol{C}$, and s the smaller vertex set [27]. We obtain an upper bound of $O(n_i n_j \Delta)$, since $m \leq n_i n_j$ and $s = \min\{n_i, n_j\} \leq \Delta$. This leads to a total running time of

$$\sum_{i \in V} \sum_{j \in V'} O(n_i n_j \Delta) = O\left(\sum_{i \in V} n_i \sum_{j \in V'} n_j \Delta \right) = O(nn'\Delta).$$

For integral edit costs bounded by K, the costs in all assignment problems are bounded by $C = K(n+n')$. Using the bipartite matching algorithm by Goldberg et al. [13] we obtain a total running time of $O(nn'\sqrt{\Delta}\log(\Delta C))$.

4.4 Deriving an Edit Path

We obtain a cost matrix for computing a minimum assignment between the vertices of the two graphs, based on the SDTED between their neighborhood trees. From the assignment an upper bound for the GED is derived yielding our approximation. For this, we follow the methodology described in the initial publication using bipartite graph matching [28], also see Sect. 2. The mapping obtained from the vertex assignment via their neighborhood trees represents a suboptimal edit path. As the overall path derived from this assignment is not guaranteed to be optimal, the cost of the derived path must be greater or equal to the cost of an optimal one. Therefore, our approximation, just like *BGM*, is an upper bound for the GED.

4.5 Theoretical Complexity

We investigate the theoretical complexity of our approach, which is dominated by the time needed for creating the cost matrix $\boldsymbol{C}$.

Theorem 3. *Given two graphs with n and n' vertices, m and m' edges, and maximum degree Δ, the running time for computing the cost matrix $\boldsymbol{C}$ based on the SDTED for cNTs with constant k is $O(nn'mm'\Delta)$.*

Proof. For each entry of the cost matrix the SDTED between two cNTs is computed. According to Theorem 1 the size of the cNTs is bounded by $O(m)$ and $O(m')$, respectively. This leads to a running time of $O(mm'\Delta)$ for SDTED computation with Theorem 2. Since the cost matrix has $n \cdot n'$ entries, the total running time is $O(nn'mm'\Delta)$.

In bounded-degree graphs, where Δ is constant and the number of edges is a constant multiple of the number of vertices, the running time is $O((nn')^2)$.

Table 1 compares the theoretical complexity of commonly used approximations for the GED based on the bipartite graph matching method. Only the cost to compute the cost matrix $\boldsymbol{C}$ is shown, as the subsequent steps (computing the assignment and deriving the edit path from it) do not differ between the methods. While *BGM* is theoretically the fastest, the substructure captured in the cost is not sufficient for good approximation quality in some cases. With increasing radius, *Subgraph* is much slower than the other approaches, as computing the exact GED between the subgraphs is computationally complex. Compared to the *Walk* method, our approach promises to be more efficient for graphs of bounded degree. We investigate the empirical runtime of these methods in Sect. 5.

4.6 Optimization Strategies

We describe implementation details and techniques to speed up the computation in practice. The most significant acceleration is achieved by storing and reusing the results of SDTED computations for neighborhood trees and subtrees. To this end, we use canonical string encodings of trees, which are equal for two trees if and only if the trees are isomorphic. While no polynomial-time algorithm is known for the graph canonization problem [15], canonical encodings of unlabeled trees can be computed in linear-time by classical algorithms [2], which sort siblings level-wise in a bottom-up fashion. While for unlabeled graphs sorting can be realized in linear-time using bucket sort, for arbitrary label alphabets $O(n \log n)$ time is required to order the tree unambiguously. From this, a string encoding is derived via tree traversal serving as a unique identifier. Here, this technique is used for accessing a hash-based data structure storing the SDTED between trees by using tuples of canonical encodings as key.

We implemented the tree canonization such that it generates canonical encodings of all subtrees as a byproduct, which are also cached in the same way. This means that, not only the encodings of subtrees can be reused for computing the encoding of all parents, but also the repeated computation of SDTED between subtrees can be avoided. Since Weisfeiler-Leman unfolding trees have a highly repetitive structure, they are expected to greatly benefit from caching.

5 Experimental Evaluation

We compare our proposed technique to state-of-the-art approaches regarding approximation quality and runtime. We address the following research questions:

Q1 How tight are the upper bounds of our method compared to state-of-the-art?
Q2 How do our bounds perform when taking the trade-off between bound quality and runtime into account?
Q3 How much of a speed-up is gained by our caching strategy?

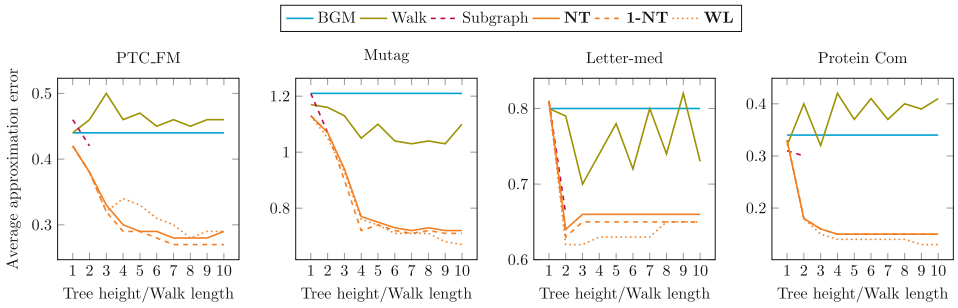


Fig. 3. Average relative approximation error of newly proposed (bold) and state-of-the art methods regarding the exact GED on the different datasets.

Methods. We compare our approach to the state-of-the-art bipartite graph matching methods *BGM* [28], its extensions using bags of walks (denoted by *Walks* or *W*) [12] and subgraphs (denoted by *Subgraph* or *SG*) [8]. The competitors employ the same framework as our approach, but offer different trade-offs between quality and running time. Other approximation algorithms are typically orders of magnitude slower. In addition, we used the state-of-the-art exact GED method *BSS_GED* [9]. For our method, we compare the assignments under the SDTED using traditional unfolding trees *WL* and neighborhood trees, investigating both 0-NTs (*NT*) and 1-NTs (1-*NT*), and varying the height parameter for all trees. We employ the optimizations described in Sect. 4.6 and apply compression to all trees generated. The parameter w to weight down the edit costs of distant nodes in the SDTED computation, was set to 0.5 for all datasets, yielding promising results in preliminary experiments. Since the exact approach can handle uniform edit costs only, we use these in our experiments.

Implementation. We implemented our methods, as well as *BGM* and its extensions, in Java[1], following the implementations presented by the authors. We used the C++ implementation of *BSS_GED* provided by the authors. All experiments were conducted on an Intel Xeon Gold 6130 machine at 2.1 GHz with 96 GB RAM. We report average execution time over 5 runs.

Datasets. We tested all methods on a wide range of real-world datasets from the TUDataset [22]. These datasets are widely used for graph similarity computation and have discrete vertex and edge labels. Attributes, if present, were removed

[1] Our implementation is available at https://github.com/frareba/NeighborhoodTrees.

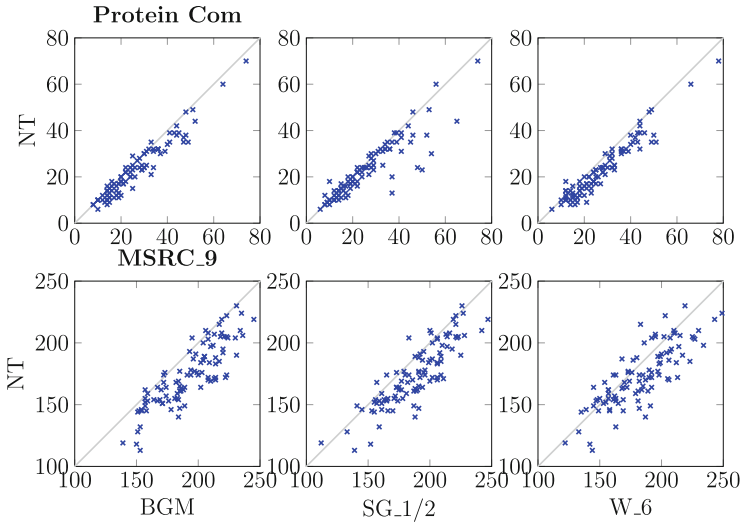


Fig. 4. Approximated GED of *NT_ 10* in comparison to state-of-the-art methods on *Protein Com* and *MSRC_ 9*. Methods are denoted with variant_parameter.

prior to the experiments since not all methods support them. Additionally, we sampled 1000 graphs from the dataset *Protein Com* [35].

Results. *Q1: Bound Quality.* Tight bounds are crucial, particularly when the exact GED is small. In this section, we compare our new approaches with state-of-the-art bipartite graph matching. We first investigate how many iterations are needed for the different tree structures to achieve the best accuracy.

For a dataset $\mathcal{D} = \{G_1, G_2, \ldots, G_N\}$ we select $n = 100$ pairs (G_i, G_j) with $i \neq j$ uniformly at random to obtain a set $\mathcal{P} = \{(G_1, H_1), (G_2, H_2), \ldots, (G_n, H_n)\} \subseteq \mathcal{D}^2$. For each pair we compute the exact GED $d_i = \text{GED}(G_i, H_i)$ and the result of the approximation method $\hat{d}_i$. We consider the average relative error for this set of pairs, i.e., $R = \frac{1}{n} \sum_{i=1}^{n} \frac{|d_i - \hat{d}_i|}{d_i}$. The average relative error of the different methods is shown in Fig. 3. Since *BGM* does not have any parameter to adjust the tree height, walk length or subgraph radius, the approximation error is constant and is used as a baseline. While our methods gain a huge advantage initially in all datasets, the approximations do not get much better after a certain tree height. An explanation for this is that many graphs are already fully explored by NTs of small height. The *Walk* method performed sometimes worse and sometimes better than the baseline *BGM*, while always performing worse than our approach. Additionally, the accuracy varies unpredictably with increasing walk length, often even leading to degraded accuracy. The *Subgraph* method could only be tested with a subgraph radius of 1 and 2 due to the huge increase in runtime (see Fig. 5). The results were comparable to or worse than our method. It can be seen, that 1-NTs are only slightly better than 0-NTs in some cases, and that the performance of *WL* is also comparable to *NT*.

We compare NT with height 10 directly to the state-of-the-art methods on an instance level in Fig. 4. Points below the gray line indicate a better approximation by our approach. Since almost all points are below this line, we can conclude, that our method performs generally better, even on datasets with larger graphs. Especially, when the (approximated) distance is small, NT performs much better in many cases. A high accuracy for small distances is very important, e.g., in k-nearest neighbors search, which is commonly used in data mining tasks such as k-nn classification.

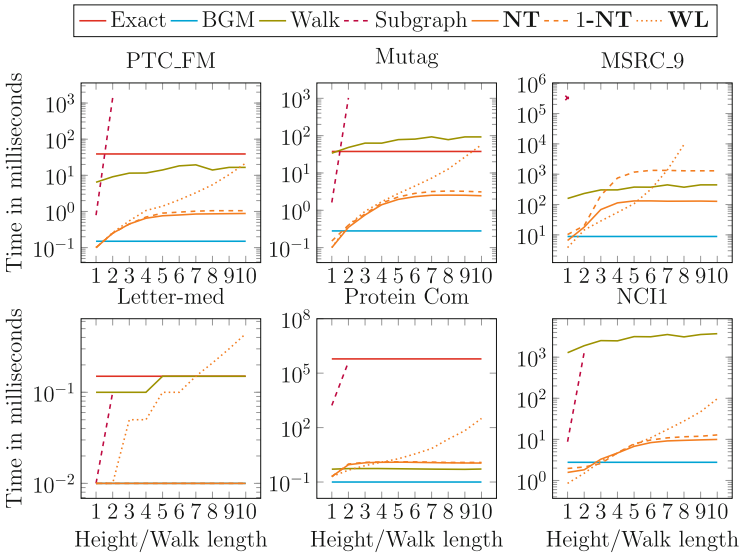


Fig. 5. Average time for computing the distance of two graphs (proposed in bold).

Q2: Runtime. There is typically a trade-off between runtime and accuracy. In this section, we evaluate our proposed approaches in terms of runtime. Figure 5 shows the average runtime for computing the distance between two graphs using the different methods. Exact computation is much slower than most approximations and shown as a baseline (for the datasets *MSRC_9* and *NCI1* no values are given, since the graphs are already too large for this method). Given that *BGM* only computes one larger assignment (along with a small assignment for each vertex pair), it is expected to be faster than our proposed methods. In practice, the runtime difference between *BGM* and $NT/1\text{-}NT$ is quite small, while our *WL* variant is much slower. Since the unfolding trees in the *WL* variant contain many redundant vertices and grow fast in size with increasing height, this aligns with our expectations. For *MSRC_9* and *NCI1* the *WL* variant is slightly faster than $NT/1\text{-}NT$ for lower heights h. This can be explained by the impact of our caching strategy, since unfolding trees contain many isomorphic subtrees allowing to skip SDTED computation whenever they reoccur. The plateauing of runtime

for *NT* and 1-*NT* can be explained by the bounded size of neighborhood trees. Once every edge in the graph is explored, the tree ceases to grow. However, *Subgraph* becomes very slow with increasing subgraph radius, to the extend where exact GED computation is faster than the approximate method. Consequently, we did not test subgraph radii of 3 and larger. The runtime of *Walk* varies a lot between the datasets relative to the other methods. Seemingly, *Walk* is fastest when the number of distinct labels is large, due to the small number of matching node pairs in the method's product graph. While *Walk* was the fastest method other than *BGM* for the Protein Com dataset, it is important to note, that it performed worse than the baseline *BGM*. Interestingly, 1-*NT* performs much worse on dataset *MSRC_9* than *NT*, while on the other datasets, there was not much of a difference between them. This could be due to the higher average vertex degree in this dataset, since 1-*NT* allows for some redundancy in the tree (by allowing vertices to appear in two consecutive levels). Given the minimal accuracy gain from using 1-*NT* over *NT*, suggests that 0-NTs are sufficient for representing the neighborhood of a vertex accurately and efficiently.

Q3: Caching. We evaluate the speed-up in runtime gained by our caching strategy for the different tree variants. For a fair comparison, the cached values are not stored for the whole dataset, but only during pairwise computation. Keeping the values for all or multiple computations may provide additional speed-up at the cost of memory, whenever subtree pairs reoccur for different graph pairs. Figure 6 shows the average runtime for computing our approximations with and without using our caching strategy and varying tree height. Since the runtime for the uncached *WL* increased very quickly with larger height, only values up to a tree height of 5 are given. On almost all datasets we can see a huge runtime benefit for all methods. Because the size of the *WL* trees is not limited by the graph size and grows quickly, *WL* benefits the most from caching. The other methods benefit more on datasets with higher average degree or larger diameter. A reason for this might be the higher number of redundant computations occurring in these graphs, which can be avoided by our caching strategy.

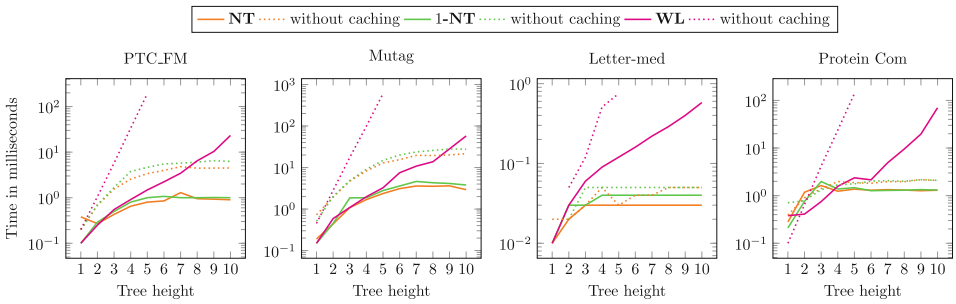


Fig. 6. Average time for computing the distance with/without caching.

Discussion. While our method provides tighter bounds for the GED, it is only slightly slower than state-of-the-art BGM. Especially for graphs with small diameter, a small tree height is often sufficient. In contrast to bags of walks, where large walk lengths can decrease the accuracy due to "tottering" [12], the accuracy of our method increases with larger tree height. This indicates, that a good choice for the tree height can easily be found, as it trades accuracy for runtime.

6 Conclusions

We introduced a new approximation for the GED based on bipartite graph matching. Using neighborhood trees and their SDTED for vertex assignment, our approach takes complex structural information around the vertices into account, without performing expensive exact GED computations for local subgraphs. As our trees can be limited in height, an individual trade-off between runtime and accuracy can be achieved. We analyzed the running time of the SDTED computation and improved it to $O(n^2\Delta)$ for two trees with n vertices and maximum degree Δ. We proposed to accelerate it, using compact compressed tree representations and tree canonization to avoid redundant computations. Experimentally we showed that our method is only slightly slower than standard bipartite graph matching, while providing great benefits regarding approximation accuracy.

Future work involves investigating other applications, such as database search, where globally storing SDTEDs seems promising, or molecular graphs for structure-based virtual screening [11], where the improved approximation quality with little overhead could be especially beneficial with larger molecules [31]. The Weisfeiler-Leman algorithm is fundamental for graph learning methods such as graph kernels and GNNs [21]. Replacing unfolding trees with compact neighborhood trees holds potential across many applications in this domain.

Acknowledgments. This work was supported by the Vienna Science and Technology Fund (WWTF) [10.47379/VRG19009].

References

1. Abu-Aisheh, Z., Raveaux, R., Ramel, J.Y., Martineau, P.: An exact graph edit distance algorithm for solving pattern recognition problems. In: ICPRAM (2015)
2. Aho, A.V., Hopcroft, J.E.: The Design and Analysis of Computer Algorithms. Addison-Wesley Longman Publishing Co., Inc., Boston (1974)
3. Bai, Y., Ding, H., Bian, S., Chen, T., Sun, Y., Wang, W.: SimGNN: a neural network approach to fast graph similarity computation. In: ACM WSDM (2019)
4. Blumenthal, D.B., Boria, N., Gamper, J., Bougleux, S., Brun, L.: Comparing heuristics for graph edit distance computation. VLDB J. **29**(1), 419–458 (2020)
5. Blumenthal, D.B., Gamper, J.: Improved lower bounds for graph edit distance. IEEE Trans. Knowl. Data Eng. **30**(3), 503–516 (2018)
6. Blumenthal, D.B., Gamper, J.: On the exact computation of the graph edit distance. Pattern Recogn. Lett. **134**, 46–57 (2020)

7. Bougleux, S., Gaüzère, B., Blumenthal, D.B., Brun, L.: Fast linear sum assignment with error-correction and no cost constraints. Pattern Recognit. Lett. **134**, 37–45 (2020)
8. Carletti, V., Gaüzère, B., Brun, L., Vento, M.: Approximate graph edit distance computation combining bipartite matching and exact neighborhood substructure distance. GbRPR (2015)
9. Chen, X., Huo, H., Huan, J., Vitter, J.S.: An efficient algorithm for graph edit distance computation. Knowl.-Based Syst. **163**, 762–775 (2019)
10. Foggia, P., Percannella, G., Vento, M.: Graph matching and learning in pattern recognition in the last 10 years. IJPRAI **28**, 1450001 (2014)
11. Garcia-Hernandez, C., Fernández, A., Serratosa, F.: Ligand-based virtual screening using graph edit distance as molecular similarity measure. J. Chem. Inform. Model. **59**(4), 1410–1421 (2019)
12. Gaüzère, B., Bougleux, S., Riesen, K., Brun, L.: Approximate graph edit distance guided by bipartite matching of bags of walks. In: Fränti, P., Brown, G., Loog, M., Escolano, F., Pelillo, M. (eds.) S+SSPR 2014. LNCS, vol. 8621, pp. 73–82. Springer, Heidelberg (2014). https://doi.org/10.1007/978-3-662-44415-3_8
13. Goldberg, A.V., Hed, S., Kaplan, H., Tarjan, R.E.: Minimum-cost flows in unit-capacity networks. Theory Comput. Syst. **61**(4), 987–1010 (2017)
14. Gouda, K., Hassaan, M.: CSI_GED: an efficient approach for graph edit similarity computation. In: International Conference on Data Engineering, ICDE (2016)
15. Grohe, M., Schweitzer, P.: The graph isomorphism problem. Commun. ACM **63**(11), 128–134 (2020)
16. Jonker, R., Volgenant, A.: A shortest augmenting path algorithm for dense and sparse linear assignment problems. Computing **38**, 325–340 (2005)
17. Kriege, N.M., Giscard, P., Bause, F., Wilson, R.C.: Computing optimal assignments in linear time for approximate graph matching. In: ICDM (2019)
18. Kriege, N.M., Giscard, P.L., Wilson, R.C.: On valid optimal assignment kernels and applications to graph classification. In: NIPS 2016 (2016)
19. Kriege, N.M., Johansson, F.D., Morris, C.: A survey on graph kernels. Appl. Netw. Sci. **5**, 1–42 (2020)
20. Lerouge, J., Abu-Aisheh, Z., Raveaux, R., Héroux, P., Adam, S.: New binary linear programming formulation to compute the graph edit distance. Pattern Recognit. **72**, 254–265 (2017)
21. Morris, C., Fey, M., Kriege, N.M.: The power of the Weisfeiler-Leman algorithm for machine learning with graphs. In: IJCAI 2021, pp. 4543–4550 (2021)
22. Morris, C., Kriege, N.M., Bause, F., Kersting, K., Mutzel, P., Neumann, M.: Tudataset: a collection of benchmark datasets for learning with graphs. GRL+ (2020)
23. Morris, C., et al.: Weisfeiler and Leman go machine learning: the story so far. CoRR (2021)
24. Morris, C., et al.: Weisfeiler and Leman go neural: higher-order graph neural networks. In: AAAI (2019)
25. Munkres, J.R.: Algorithms for the assignment and transportation problems. J. Soc. Ind. Appl. Math. **5**(1), 32–38 (1957)
26. Piao, C., Xu, T., Sun, X., Rong, Y., Zhao, K., Cheng, H.: Computing graph edit distance via neural graph matching. Proc. VLDB Endow. **16**(8), 1817–1829 (2023)
27. Ramshaw, L., Tarjan, R.E.: On minimum-cost assignments in unbalanced bipartite graphs. Technical report, HPL-2012-40, HP Laboratories (2012)
28. Riesen, K., Bunke, H.: Approximate graph edit distance computation by means of bipartite graph matching. Image Vision Comput. **27**(7), 950–959 (2009)

29. Schenker, A., Kandel, A., Bunke, H., Last, M.: Graph-Theoretic Techniques for Web Content Mining. Series in Machine Perception and AI (2005)
30. Schulz, T.H., Horváth, T., Welke, P., Wrobel, S.: A generalized Weisfeiler-Lehman graph kernel. Mach. Learn. **111**(7), 2601–2629 (2022)
31. Sengupta, S., Mehta, G.: Macrocyclization via C-H functionalization: a new paradigm in macrocycle synthesis. Org. Biomol. Chem. **18**(10), 1851–1876 (2020)
32. Serratosa, F.: Speeding up fast bipartite graph matching through a new cost matrix. Int. J. Pattern Recognit. Artif. Intell. **29**(2), 1550010:1–1550010:17 (2015)
33. Shervashidze, N., Schweitzer, P., van Leeuwen, E.J., Mehlhorn, K., Borgwardt, K.M.: Weisfeiler-Lehman graph kernels. JMLR **12**(77), 2539–2561 (2011)
34. Stauffer, M., Tschachtli, T., Fischer, A., Riesen, K.: A survey on applications of bipartite graph edit distance. GbRPR (2017)
35. Stöcker, B.K., Schäfer, T., Mutzel, P., Köster, J., Kriege, N., Rahmann, S.: Protein complex similarity based on Weisfeiler-Lehman labeling. In: Amato, G., Gennaro, C., Oria, V., Radovanović, M. (eds.) SISAP 2019. LNCS, vol. 11807, pp. 308–322. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-32047-8_27
36. Xu, K., Hu, W., Leskovec, J., Jegelka, S.: How powerful are graph neural networks? In: 7th International Conference on Learning Representations, ICLR 2019 (2019)
37. Zeng, Z., Tung, A.K.H., Wang, J., Feng, J., Zhou, L.: Comparing stars: on approximating graph edit distance. Proc. VLDB Endow. **2**(1), 25–36 (2009)

A.4 Gradual Weisfeiler-Leman: Slow and Steady Wins the Race

Authors Franka Bause and Nils M. Kriege

Publication Venue Published in *Proceedings of the First Learning on Graphs Conference*, PMLR 198 (2022).

DOI/URL https://proceedings.mlr.press/v198/bause22a.html

Authors Contributions

Franka Bause developed the theoretical foundation of renep functions and the proofs in the paper, implemented the algorithm, conducted the experimental evaluation, and drafted, wrote, and revised the manuscript.

Nils M. Kriege initiated the project with the idea of a gradual version of the Weisfeiler-Leman algorithm, supervised the project, gave valuable feedback, and helped to revise the manuscript.

Reference in Thesis [BK22]

Abstract The classical Weisfeiler-Leman algorithm aka color refinement is fundamental for graph learning with kernels and neural networks. Originally developed for graph isomorphism testing, the algorithm iteratively refines vertex colors. On many datasets, the stable coloring is reached after a few iterations and the optimal number of iterations for machine learning tasks is typically even lower. This suggests that the colors diverge too fast, defining a similarity that is too coarse. We generalize the concept of color refinement and propose a framework for gradual neighborhood refinement, which allows a slower convergence to the stable coloring and thus provides a more fine-grained refinement hierarchy and vertex similarity. We assign new colors by clustering vertex neighborhoods, replacing the original injective color assignment function. Our approach is used to derive new variants of existing graph kernels and to approximate the graph edit distance via optimal assignments regarding vertex similarity. We show that in both tasks, our method outperforms the original color refinement with only a moderate increase in running time advancing the state of the art.

Gradual Weisfeiler-Leman: Slow and Steady Wins the Race

Franka Bause[1,2] **Nils M. Kriege**[1,3]
[1]Faculty of Computer Science, University of Vienna, Vienna, Austria
[2]UniVie Doctoral School Computer Science, University of Vienna, Vienna, Austria
[3]Research Network Data Science, University of Vienna, Vienna, Austria
{franka.bause, nils.kriege}@univie.ac.at

Abstract

The classical Weisfeiler-Leman algorithm aka color refinement is fundamental for graph learning with kernels and neural networks. Originally developed for graph isomorphism testing, the algorithm iteratively refines vertex colors. On many datasets, the stable coloring is reached after a few iterations and the optimal number of iterations for machine learning tasks is typically even lower. This suggests that the colors diverge too fast, defining a similarity that is too coarse. We generalize the concept of color refinement and propose a framework for gradual neighborhood refinement, which allows a slower convergence to the stable coloring and thus provides a more fine-grained refinement hierarchy and vertex similarity. We assign new colors by clustering vertex neighborhoods, replacing the original injective color assignment function. Our approach is used to derive new variants of existing graph kernels and to approximate the graph edit distance via optimal assignments regarding vertex similarity. We show that in both tasks, our method outperforms the original color refinement with only a moderate increase in running time advancing the state of the art.

1 Introduction

The (1-dimensional) Weisfeiler-Leman algorithm, also referred to as *color refinement*, iteratively refines vertex colors by encoding colors of neighbors and was originally developed as a heuristic for the graph isomorphism problem. Although it cannot distinguish some non-isomorphic graph pairs, for example strongly regular graphs, it succeeds in many cases. It is widely used as a sub-routine in isomorphism algorithms today to reduce ambiguities that have to be resolved by backtracking search [1]. It has also gained high popularity in graph learning, where the technique is used to define graph kernels [2–5] and to formalize the expressivity of graph neural networks, see the recent surveys [6, 7]. Graph kernels based on Weisfeiler-Leman refinement provide remarkable predictive performance while being computationally highly efficient. The original Weisfeiler-Leman subtree kernel [2] and its variants and extensions, e.g., [3–5], provide state-of-the-art classification accuracy on many datasets and are widely used baselines. The update scheme of the Weisfeiler-Leman algorithm is similar to the idea of neighborhood aggregation in graph neural networks (GNNs). It has been shown that (i) the expressive power of GNNs is limited by the Weisfeiler-Leman algorithm, and (ii) that GNN architectures exist that reach this expressive power [8, 9].

As a consequence of its original application, the Weisfeiler-Leman algorithm assigns discrete colors and does not distinguish minor and major differences in vertex neighborhoods. Most Weisfeiler-Leman graph kernels match vertex colors of the first few refinement steps by equality, which can be considered too rigid, since these colors encode complex neighborhood structures. In machine learning tasks, a more fine-grained differentiation appears promising. Real-world data is often noisy leading to small differences in vertex degree. Such differences get picked up by the refinement strategy of the Weisfeiler-Leman algorithm and cannot be distinguished from significant differences.

We address this problem by providing a different approach to the refinement step of the Weisfeiler-Leman algorithm: We replace the injective relabeling function with a non-injective one to gain a more

F. Bause et al., Gradual Weisfeiler-Leman: Slow and Steady Wins the Race. *Proceedings of the First Learning on Graphs Conference (LoG 2022)*, PMLR 198, Virtual Event, December 9–12, 2022.

gradual refinement of colors. This allows obtaining a finer vertex similarity measure, distinguishing between large and small changes in vertex neighborhoods with increasing radius. We characterize the set of functions that, while not necessarily injective, guarantee that the stable coloring of the original Weisfeiler-Leman algorithm is reached after a possibly higher number of iterations. Thus, our approach preserves the expressive power of the Weisfeiler-Leman algorithm. We discuss a realization of such a function and use k-means clustering in our experimental evaluation as an exemplary one.

Our Contribution.

1. We propose refining, neighborhood preserving (*renep*) functions, which generalize the concept of color refinement. This family of functions leads to the coarsest stable coloring while only incorporating direct neighborhoods.
2. We show the connections of our approach to the original Weisfeiler-Leman algorithm, as well as other vertex refinement strategies.
3. We propose two new graph kernels based on renep functions, which outperform state-of-the-art kernels on synthetic and real-world datasets, with only a moderate increase in running time.
4. We apply our new approach for approximating the graph edit distance via bipartite graph matching and show that it outperforms state-of-the-art heuristics.

2 Related Work

Various graph kernels based on Weisfeiler-Leman refinement have been proposed [2–5, 10]. Recent comprehensive experimental evaluations confirm their high classification accuracy on many real-world datasets [11, 12]. In both studies, the classical Weisfeiler-Leman subtree kernel [2] is shown to provide a competitive baseline. Variants based on optimal assignments [5, 10] improve the classification accuracy on some datasets and achieve excellent results overall.

Most Weisfeiler-Leman based approaches implicitly match colors by equality, which can be considered too rigid, since colors encode unfolding trees representing complex neighborhood structures. Some recent works address this problem: Yanardag and Vishwanathan [13] introduced similarities between colors using techniques inspired by natural language processing, which were subsequently refined by Narayanan et al. [14]. Schulz et al. [15] define a distance function between colors by comparing the associated unfolding trees using a tree edit distance. Based on this distance, the colors are clustered to obtain a new graph kernel. Although the tree edit distance is polynomial-time computable, the running time of the algorithm is very high. A kernel based on the Wasserstein distance of sets of unfolding trees was proposed by Fang et al. [16]. The vertices of the graphs are embedded into ℓ_1 space using an approximation of the tree edit distance between their unfolding trees. A graph can then be seen as a distribution over those embeddings. While the function proposed is not guaranteed to be positive semidefinite, the method showed results similar to and, in some cases, exceeding state-of-the-art techniques. The running time, however, is still very high and the method is only feasible for unfolding trees of small height. These approaches define similarities between Weisfeiler-Leman colors and the associated unfolding trees. Our approach, in contrast, alters the Weisfeiler-Leman refinement procedure itself and does not rely on the computationally expensive matching of unfolding trees.

Some graph kernels use a neighborhood aggregation scheme similar in spirit to the Weisfeiler-Leman algorithm with a non-injective relabeling function. The neighborhood hash kernel [17] represents labels by bit-vectors and uses logical operations and hashing to encode neighborhoods efficiently. Shervashidze [18, Section 3.7.3.2] proposed to compress neighborhood label histograms using locality sensitive hashing allowing collisions of similar neighborhoods. Propagation kernels [19] provide a generic framework to obtain graph kernels from (neighborhood) propagation schemes by comparing label distributions after every propagation step. In contrast to these methods, our approach uses a data-dependent non-injective relabel function and guarantees that the expressive power of the Weisfeiler-Leman algorithm is reached.

Several authors proposed techniques supporting graphs with continuous attributes directly based on or inspired by the Weisfeiler-Leman algorithm. Propagation kernels were adapted to this setting by using a hash function to map continuous attributes to discrete labels [19]. The hash graph kernel framework [20] repeatedly performs such a discretization using random hash functions, applies a kernel for graphs with discrete labels to each resulting graph and combines their feature vectors. It

obtains high classification accuracies when combined with the Weisfeiler-Leman subtree kernel. The Wasserstein Weisfeiler-Leman graph kernel [3] establishes node matchings similar to the Weisfeiler-Leman optimal assignment kernel [5, 7]. The authors support continuous attributes by replacing discrete colors with real-valued vectors without guaranteeing that the resulting function is positive semidefinite. Finally, graph neural networks can be viewed as a neural version of the Weisfeiler-Leman algorithm, where continuous feature vectors replace colors, and neural networks are used to aggregate over local node neighborhoods [7–9]. None of these methods explicitly aims to slow down the Weisfeiler-Leman refinement process.

3 Preliminaries

In this section we provide the definitions necessary to understand our new vertex refinement algorithm. We first give a short introduction to graphs and the original Weisfeiler-Leman algorithm, before we cover graph kernels.

Graph Theory. A *graph* $G = (V, E, \mu, \nu)$ consists of a set of vertices V, denoted by $V(G)$, a set of edges $E(G) = E \subseteq \binom{V}{2}$ between the vertices, a labeling function for the vertices $\mu\colon V \to L$, and a labeling function for the edges $\nu\colon E \to L$. The set L contains categorical labels, which can be represented as natural numbers. We discuss only undirected graphs and denote an edge between u and v by uv. The set of neighbors of a vertex $v \in V$ is denoted by $N(v) = \{u \mid uv \in E\}$. A *(rooted) tree* T is a simple (no self-loops or multi-edges), connected graph without cycles and with a designated node r called *root*. A tree T' is a *subtree* of a tree T, denoted by $T' \subseteq T$, iff $V(T') \subseteq V(T)$. The root of T' is the node closest to the root in T.

A vertex *coloring* $c\colon V(G) \to \mathbb{N}_0$ of a graph G is a function assigning each vertex a color. For vertices with $c(u) = c(v)$ we also write $u \approx_c v$. A coloring π on a set S is a *refinement* of (or *refines*) a coloring π', iff $s_1 \approx_\pi s_2 \Rightarrow s_1 \approx_{\pi'} s_2$ for all s_1, s_2 in S. We denote this by $\pi \preccurlyeq \pi'$ and write $\pi \equiv \pi'$ if $\pi \preccurlyeq \pi'$ and $\pi' \preccurlyeq \pi$. If $\pi \preccurlyeq \pi'$ and $\pi \not\equiv \pi'$, we say that π is a *strict refinement* of π', written $\pi \prec \pi'$. The refinement relation defines a partial ordering on the colorings.

Color Hierarchy. We consider a sequence of vertex colorings $(\pi_0, \pi_1, \dots, \pi_h)$ with $\pi_h \preccurlyeq \dots \preccurlyeq \pi_0$ and assume that for $i \neq j$ the colors assigned by π_i and π_j are distinct. We can interpret such a sequence of colorings as a *color hierarchy*, i.e., a tree $\mathcal{T}_h$ that contains a node for each color $c \in \{\pi_i(v) \mid i \in \{0, \dots, h\} \wedge v \in V(G)\}$ and an edge (c, d) iff $\exists v \in V(G)\colon \pi_i(v) = c \wedge \pi_{i+1}(v) = d$. We associate each tree node with the set of vertices of G having that color.[1] Here, we assume that the initial coloring is uniform. If this is not the case, we add an artificial root node and connect it to the initial colors. Likewise we insert the coloring $\pi_0 = \{V(G)\}$ as first element in the sequence of vertex colorings. An example color hierarchy is given in Figure 1.

Using this color hierarchy we can derive multiple colorings on the vertices: Choosing exactly one color on every path from the leaves to the root (or only the root), always leads to a valid coloring. The finest coloring is induced by the colors representing the leaves of the tree. Given a color hierarchy T, we denote this coloring (which is equal to π_h) by π_T.

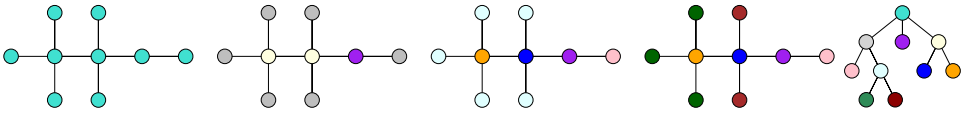
Weisfeiler-Leman Color Refinement. The 1-dimensional Weisfeiler-Leman (WL) algorithm or color refinement [21, 22] starts with a coloring c_0, where all vertices have a color representing their label (or a uniform coloring in case of unlabeled vertices). In iteration i, the coloring c_i is obtained by assigning each vertex v in $V(G)$ a new color according to the colors of its neighbors, i.e.,

$$c_{i+1}(v) = z\left(c_i(v), \{\!\{c_i(u) \mid u \in N(v)\}\!\}\right),$$

where $z\colon \mathbb{N}_0 \times \mathbb{N}_0^{\mathbb{N}_0} \to \mathbb{N}_0$ is an injective function. Figure 1 depicts the first iterations of the algorithm for an example graph.

After enough iterations the number of different colors will no longer change and this resulting coloring is called the *coarsest stable coloring*. The coarsest stable coloring is unique and always reached after at most $|V(G)| - 1$ iterations. This trivial upper bound on the number of iterations is tight [23]. In practice, however, Weisfeiler-Leman refinement converges much faster (see Appendix C).

[1] Here, we consider a color hierarchy for a single graph. However, given corresponding colorings on a set of graphs, a color hierarchy can be generated in the same way. In an inductive setting, where no fixed set of graphs is given, the color hierarchy contains all colors, that can be produced by the corresponding coloring algorithm.



(a) Initial colors **(b)** Iteration 1 **(c)** Iteration 2 **(d)** Iteration 3 **(e)** Hierarchy

Figure 1: Initial coloring and results of the first three iterations of the Weisfeiler-Leman algorithm. To use less colors for this example, vertices with a unique color do not get a new color. The color hierarchy shows the development of the colors over the refinement iterations.

Graph Kernels and the Weisfeiler-Leman Subtree Kernel. A *kernel* on X is a function $k\colon X \times X \to \mathbb{R}$, so that there exist a Hilbert space $\mathcal{H}$ and a mapping $\phi\colon X \to \mathcal{H}$ with $k(x,y) = \langle \phi(x), \phi(y) \rangle$ for all x, y in X, where $\langle \cdot, \cdot \rangle$ is the inner product of $\mathcal{H}$. A *graph kernel* is a kernel on graphs, i.e., X is the set of all graphs.

The Weisfeiler-Leman subtree kernel [2] with height h is defined as

$$k_{ST}^{h}(G_1, G_2) = \sum_{i=0}^{h} \sum_{u \in V(G_1)} \sum_{v \in V(G_2)} \delta(c_i(u), c_i(v)), \tag{1}$$

where δ is the Dirac kernel (1, iff $c_i(u)$ and $c_i(v)$ are equal, and 0 otherwise). It counts the number of vertices with common colors in the two graphs up to the given bound on the number of Weisfeiler-Leman iterations.

4 Gradual Weisfeiler-Leman Refinement

As a different approach to the refinement step of the Weisfeiler-Leman algorithm, we essentially replace the injective relabeling function with a non-injective one. We do this by allowing vertices with differing neighbor color multisets to be assigned the same color under some conditions. Through this, the number of colors per iteration can be limited, allowing to obtain a more gradual refinement of colors. To reach the same stable coloring as the original Weisfeiler-Leman algorithm, the function has to assure that vertices with differing colors in one iteration will get differing colors in future iterations and that in each iteration at least one color is split up, if possible.

We first define the property necessary to reach the stable coloring of the original Weisfeiler-Leman algorithm and discuss connections to the original as well as other vertex refinement algorithms. Then we provide a realization of such a function by means of clustering, which is used in our experimental evaluation. Figure 2 illustrates our idea. It depicts the initial coloring, the result of the first iteration of WL and a possible result of the first iteration of the gradual Weisfeiler-Leman refinement (GWL), when restricting the maximum number of new colors to two by clustering the neighbor color multisets.

Update Functions. Using the same approach as the Weisfeiler-Leman algorithm, the color of a vertex is updated iteratively according to the colors of its neighbors. Let $\mathcal{T}_i$ denote a color hierarchy belonging to G and $n_i(v) = \{\!\{\pi_{\mathcal{T}_i}(x) \mid x \in N(v)\}\!\}$ the neighbor color multiset of v in iteration i. We use a similar update strategy, but generalize it using a special type of function:

$$\forall v \in V(G)\colon c_{i+1}(v) = \pi_{\mathcal{T}_{i+1}}(v), \text{with } \mathcal{T}_{i+1} = f(G, \mathcal{T}_i),$$

where f is a refining, neighborhood preserving function.

A *refining, neighborhood preserving (renep)* function f maps a pair $(G, \mathcal{T}_i)$ to a tree $\mathcal{T}_{i+1}$, such that

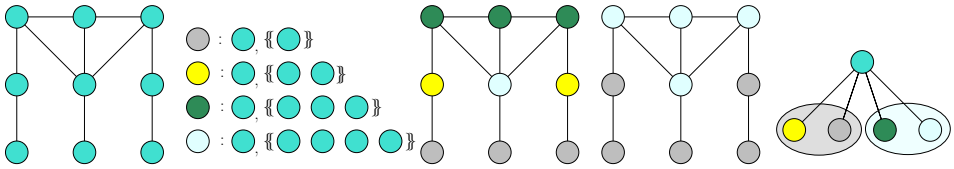
Condition 1. $\mathcal{T}_i \subseteq \mathcal{T}_{i+1}$

Condition 2. $\mathcal{T}_i = \mathcal{T}_{i+1}$, *iff* $\forall v, w \in V(G) : v \approx_{\pi_{\mathcal{T}_i}} w \Rightarrow n_i(v) = n_i(w)$

Condition 3. $\mathcal{T}_i \subsetneq \mathcal{T}_{i+1} \Rightarrow \pi_{\mathcal{T}_{i+1}} \prec \pi_{\mathcal{T}_i}$

Condition 4. $\forall v, w \in V(G)\colon (v \approx_{\pi_{\mathcal{T}_i}} w \wedge n_i(v) = n_i(w)) \Rightarrow v \approx_{\pi_{\mathcal{T}_{i+1}}} w$

The conditions assure, that the coloring $\pi_{\mathcal{T}_{i+1}}$ is a strict refinement of $\pi_{\mathcal{T}_i}$, if there exists a strict refinement: Condition 1 assures that the new coloring is a refinement of the old one. Condition 2 assures that the tree (and in turn the coloring) only stays the same, iff the stable coloring is reached,



(a) Initial colors **(b)** Neighbor colors **(c)** Result of WL **(d)** Result of GWL **(e)** Color hierarchy

Figure 2: Initial coloring and results of the first iteration using WL and GWL refinement. We assume that the update function of GWL is a clustering algorithm, producing two clusters per old color. Vertices colored gray and yellow by WL are put into the same cluster, as well as green and light blue ones, as their neighbor color multisets only differ by one element each.

while Condition 3 assures that, if the trees are not equal, $\pi_{\mathcal{T}_{i+1}}$ is a strict refinement of $\pi_{\mathcal{T}_i}$. Without this condition it would be possible to obtain a tree, that fulfills Condition 1 but does not strictly refine the coloring (for example by adding one child to each leaf). Condition 4 assures that vertices, that are indistinguishable regarding their color and their neighbor color multiset, get the same color (as in the original Weisfeiler-Leman algorithm).

We call this new approach *gradual Weisfeiler-Leman refinement* (GWL refinement). Since f is a renep function, it is assured that at least one color is split into at least two new colors, if the stable coloring is not yet reached. This property and its implications are explored in the following section.

Usually, the refinement is computed simultaneously for multiple graphs. This can be realized by using the disjoint union of all graphs as input. Note that this will have an influence on the function f, since refinements might differ based on the vertices involved. This is a typical case of transductive learning, because the algorithm has to run on all graphs and if a new graph is encountered, the algorithm has to run again on the enlarged graph. To counteract this, one could only run the algorithm on the training set and keep track of the coloring choices. Then for new graphs, the vertices are colored with the most similar colors.

4.1 Equivalence of the Stable Colorings

The gradual color refinement will never assign two vertices the same color, if their colors differed in the previous iteration, since we require the coloring to be a refinement of the previous one. We can show that the stable coloring obtained by GWL refinement using any renep function is equal to the unique coarsest stable coloring, which is obtained by the original Weisfeiler-Leman algorithm.

Theorem 5 ([24], Proposition 3)**.** *For every coloring π of $V(G)$, there is a unique coarsest stable coloring p that* refines *π.*

This means GWL with any renep function, should it reach a coarsest stable coloring, will reach this unique coarsest stable coloring. It remains to show that GWL will reach a coarsest stable coloring.

Theorem 6. *For all G the GWL refinement using any renep function will find the unique coarsest stable coloring of $V(G)$.*

Proof. Let $\pi_{\mathcal{T}} = \{p_1, \dots, p_n\}$ be the stable coloring obtained from GWL on the initial coloring π_0. Assume there exists another stable coloring $\pi' = \{p'_1, \dots, p'_m\}$ with $\pi_{\mathcal{T}} \prec \pi' \preccurlyeq \pi_0$, so $m < n$. Then $\exists v, w \in V(G) : (v \approx_{\pi'} w \wedge v \not\approx_{\pi_{\mathcal{T}}} w)$ and since Condition 4 applies $n(v) \neq n(w)$, which contradicts the assumption that π' is stable. □

The original Weisfeiler-Leman refinement can be realized by using the renep function with $\Leftrightarrow$ instead of $\Rightarrow$ in Condition 4. This ensures that vertices get assigned the same color, iff they previously had the same color and their neighborhood color multisets do not differ. Since this procedure splits all colors, that can be split up, it is the fastest converging possible renep function (because only direct neighborhood is considered). A trivial upper bound for the maximum number of Weisfeiler-Leman iterations needed is $\left|V(G)\right| - 1$ and there are infinitely many graphs on which this number of iterations is required for convergence [23]. We obtain the same upper bound for GWL.

Theorem 7. *The maximum number of iterations needed to reach the stable coloring using GWL refinement is $\left|V(G)\right| - 1$.*

Proof. The function we consider is a renep function. It follows that, prior to reaching the stable coloring, at least one color is split into at least two new colors in every iteration. Since vertices that had different colors at any step will also have different colors in the following iterations, the number of colors increases in every step. Hence, after at most $\left|V(G)\right| - 1$ steps, each vertex has a unique color, which is a stable coloring. □

Sequential Weisfeiler-Leman. For optimizing the running time of the Weisfeiler-Leman algorithm, sequential refinement strategies have been proposed [24–26], which lead to the same stable coloring as the original WL. Our presentation follows Berkholz et al. [24], who provide implementation details and a thorough complexity analysis. Sequential WL manages a stack containing the colors that still have to be processed. All initial colors are added to this stack. In each step, the next color c from the stack is used to refine the current coloring π (and generate a new coloring π') using the following update strategy: $\forall v, w \in V(G) \colon v \approx_{\pi'} w \Leftrightarrow \left|\{\!\{x \mid x \in N(v) \wedge \pi(x) = c\}\!\}\right| = \left|\{\!\{x \mid x \in N(w) \wedge \pi(x) = c\}\!\}\right| \wedge v \approx_{\pi} w$. Note that $\pi' \prec \pi$ is not guaranteed. For colors that are split, all new colors are added to the stack with exception of the largest color class. This is shown to be sufficient for generating the coarsest stable coloring [24].

Sequential Weisfeiler-Leman can be realized by our GWL with the restriction, that in sequential WL, some refinement operations might not produce strict refinements. We need to skip these in our approach (since renep functions have to produce strict refinements as long as the coloring is not stable). The renep function has to fulfill $\forall v, w \in V(G) \colon v \approx_{\pi_{\mathcal{T}_{i+1}}} w \Leftrightarrow \left|\{\!\{x \mid x \in N(v) \wedge \pi_{\mathcal{T}_i}(x) = c\}\!\}\right| = \left|\{\!\{x \mid x \in N(w) \wedge \pi_{\mathcal{T}_i}(x) = c\}\!\}\right| \wedge v \approx_{\pi_{\mathcal{T}_i}} w$, where c is the next color in the stack that produces a strict refinement.

4.2 Running Time

The running time of the gradual Weisfeiler-Leman refinement depends on the cost of the update function used.

Theorem 8. *The running time for the gradual Weisfeiler-Leman refinement is $O(h \cdot t_u(\left|V(G)\right|))$, where h is the number of iterations and $t_u(n)$ is the time needed to compute the renep function for n elements.*

The update function used in the original Weisfeiler-Leman refinement can be computed in time $O(\left|V(G)\right| + \left|E(G)\right|)$ in the worst-case by sorting the neighbor color multisets using bucket sort [2].

4.3 Discussion of Suitable Update Functions

The update function of the original Weisfeiler-Leman refinement provides a fast way to reach the stable coloring, but in machine learning tasks a more fine grained vertex similarity might be desirable. A suitable update function restricts the number of new colors to a manageable amount, while still fulfilling the requirements of a renep function. Clustering the neighborhood multisets of the vertices, and letting the clusters imply the new colors, is an intuitive way to restrict the number of colors per iteration and assign similar neighborhoods the same new color. We discuss how to realize a renep function using clustering.

Whether two vertices, that currently have the same color, will be assigned the same color in the next step, depends on two factors: If they have the same neighbor color multiset, they have to remain in one color group. If their neighbor color multisets differ, however, the renep function can decide to either separate them or not (provided any new colors are generated to fulfill Condition 3). We propose clustering the neighbor color multisets separately for each old color and let the clusters imply new colors. If a clustering function guarantees to produce at least two clusters for inputs with at least two distinct objects, we obtain a renep function.

Although various clustering algorithms are available, we identified k-means as a convenient choice because of its efficiency and controllability of the number of clusters. In order to apply k-means to multisets of colors, we represent them as (sparse) vectors, where each entry counts the number of neighbors with a specific color. The above method using k-means clustering with $k > 1$ satisfies the requirements of a renep function. Of course, if the number of elements to cluster is less than or equal to k, the clustering can be omitted and each element can be assigned its own cluster. The number of clusters in iteration i is bounded by $|L| \cdot k^i$, since each color can split into at most k new colors in each iteration and the initial coloring has at most $|L|$ colors.

5 Applications

The gradual Weisfeiler-Leman refinement provides a more fine-grained approach to capture vertex similarity, where two vertices are considered more similar, the longer it takes until they get assigned different colors. This makes the approach applicable not only to vertex classification, but also in graph kernels and as a vertex similarity measure for graph matching. We further describe these possible applications in the following and evaluate them against the state-of-the-art methods in Section 6.

Graph Kernels. The idea of the GWL subtree kernel is essentially the same as the Weisfeiler-Leman subtree kernel [2], but instead of using the original Weisfeiler-Leman algorithm, the GWL algorithm is used to generate the features. We use the definition given in Equation (1) replacing the Weisfeiler-Leman colorings with the coloring from the GWL algorithm. The Weisfeiler-Leman optimal assignment kernel [5] is obtained from an optimal assignment between the vertices of two graphs regarding a vertex similarity defined by a color hierarchy. We replace the Weisfeiler-Leman color hierarchy used originally by the one from our gradual refinement. We evaluate the performance of our newly proposed kernels in Section 6.

Tree Metrics for Approximating the Graph Edit Distance. The graph edit distance, a general distance measure for graphs, is usually approximated due to its complexity. Many approximations find an optimal assignment between the vertices of the two graphs and derive a (sub-optimal) edit path from this assignment. Naturally, the cost of such an edit path is an upper bound for the graph edit distance. An optimal assignment can be computed in linear time, if the underlying cost function is a tree metric [27], which means an approximation of the graph edit distance can be found in linear time, given a tree metric on the vertices. The color hierarchy, see Figure 1, produced by the original Weisfeiler-Leman refinement, as well as our gradual variant, can be interpreted as such a tree metric. Lin [27] approximated the graph edit distance as described above. We can again replace the original color hierarchy by the one produced by our gradual Weisfeiler-Leman refinement. Appendix I includes a more detailed explanation on how to approximate the graph edit distance using assignments. We evaluate this approximation of the graph edit distance regarding its accuracy in $k$nn-classification against the state-of-the-art and the original approach.

6 Experimental Evaluation

We evaluate the proposed approach regarding its applicability in graph kernels, as the gradual Weisfeiler-Leman subtree kernel (GWL) and the gradual Weisfeiler-Leman optimal assignment kernel (GWLOA), as well as its usefulness as a tree metric for approximating the graph edit distance. Specifically, we address the following research questions:

- **Q1** Can our kernels compete with state-of-the-art methods regarding classification accuracy on real-world and synthetic datasets?
- **Q2** Which refinement speed is appropriate and are there dataset-specific differences?
- **Q3** How do our kernels compare to the state-of-the-art methods in terms of running time?
- **Q4** Is the vertex similarity obtained from GWL refinement suitable for approximating the graph edit distance and solving learning problems with it?

We compare to the Weisfeiler-Leman subtree kernel (WLST) [2], the Weisfeiler-Leman optimal assignment kernel (WLOA) [5], as well as the approximation of the relaxed Weisfeiler-Leman subtree kernel (RWL*) [15], and the deep Weisfeiler-Leman kernel (DWL) [13]. We do not compare to [4], since the kernel showed results similar to the WLOA kernel. We compare the graph edit distance approximation using our tree metric GWLT to the original approach Lin [27] and state-of-the-art method BGM [28]. Our implementation is publicly available at `https://github.com/frareba/GradualWeisfeilerLeman`.

6.1 Setup

As discussed in Section 4.3 we used k-means clustering in our new approach. If for any color less than k different vectors were present in the clustering step, each distinct vector got its own cluster. We implemented our GWL, GWLOA and also the original WLST and WLOA in Java. We used the

Table 1: Average classification accuracy and standard deviation (highest accuracies marked in **bold**).

Kernel	PTC_FM	KKI	EGO-1	EGO-2	EGO-3	EGO-4
WLST	64.16 ±1.30	49.97 ±2.88	51.30 ±2.42	57.15 ±1.61	56.15 ±1.67	53.40 ±1.77
DWL	64.18 ±1.46	50.93 ±2.87	55.80 ±1.35	56.50 ±1.64	55.90 ±1.64	53.25 ±2.81
RWL*	62.43 ±1.46	46.54 ±4.03	65.60 ±2.74	70.20 ±1.36	**67.60** ±1.07	74.25 ±2.12
WLOA	62.34 ±1.39	48.72 ±4.05	55.95 ±1.11	60.30 ±2.00	54.25 ±1.35	52.30 ±2.29
GWL	62.61 ±1.94	**57.79** ±3.95	67.95 ±2.05	**73.65** ±1.86	65.45 ±1.88	**77.45** ±1.97
GWLOA	**64.58** ±1.77	47.47 ±2.41	**69.80** ±1.65	72.40 ±2.52	67.45 ±1.69	75.35 ±1.67
	COLLAB	DD	IMDB-B	MSRC_9	NCI1	REDDIT-B
WLST	78.98 ±0.22	79.00 ±0.52	72.01 ±0.80	90.13 ±0.75	85.96 ±0.18	80.81 ±0.52
DWL	78.93 ±0.18	78.92 ±0.40	72.36 ±0.56	90.50 ±0.76	85.68 ±0.18	80.83 ±0.40
RWL*	77.94 ±0.38	77.52 ±0.65	72.96 ±0.86	88.86 ±0.89	79.45 ±0.32	77.69 ±0.31
WLOA	80.81 ±0.22	**79.44** ±0.31	72.60 ±0.89	90.68 ±0.92	**86.29** ±0.13	89.40 ±0.14
GWL	80.62 ±0.33	79.00 ±0.81	**73.66** ±1.25	88.32 ±1.20	85.33 ±0.35	86.46 ±0.35
GWLOA	**81.30** ±0.29	78.49 ±0.57	72.88 ±0.79	**91.27** ±1.06	85.36 ±0.36	**89.98** ±0.34

RWL* and DWL Python implementations provided by the authors. Note that in contrast to the other approaches, the RWL* implementation uses multi-threading.

For evaluation, we used the C-SVM implementation LIBSVM [29] and report average classification accuracies obtained by 10-fold nested cross-validation repeated 10 times with random fold assignments. The parameters of the SVM and the kernels were optimized by the inner cross-validation on the current training fold using grid search. We chose $C \in \{10^{-3}, 10^{-2}, \dots, 10^{3}\}$ for the SVM, $h \in \{0, \dots, 10\}$ for WLST, WLOA, GWL and GWLOA and k-means with $k \in \{2, 4, 8, 16\}$. For RWL* we used $h \in \{1, \dots, 4\}$ and default values for the other parameters. In DWL the window size w and dimension d were set to 25, since this generally worked best out of the combinations from $d, w \in \{5, 25, 50\}$ and no defaults were provided. We used the default settings for the other parameters and again $h \in \{1, \dots, 10\}$. The running time experiments were conducted on an Intel Xeon Gold 6130 machine at 2.1 GHz with 96 GB RAM. For evaluating the approximation of the graph edit distance, we compare the 1-nn classification accuracy and used the Java implementation of Lin provided by the authors and implemented our approach GWLT, as well as BGM, also in Java for a fair comparison.

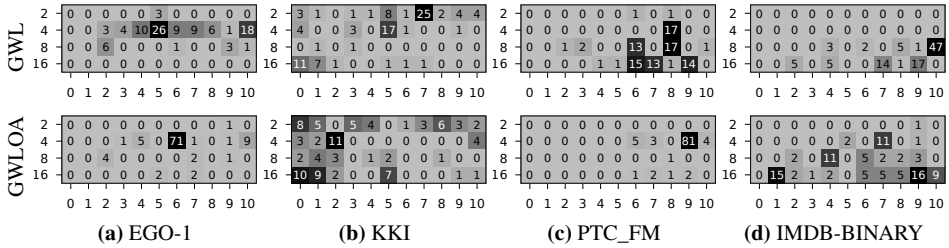
Extension to Edge Labels. The original Weisfeiler-Leman algorithm can be extended to respect edge labels by updating the colors according to $c_{i+1}(v) = z(c_i(v), \{\!\{(l(u,v), c_i(u)) \mid u \in N(v)\}\!\})$. All kernels used in the comparison use a similar strategy to incorporate edge labels if present.

Datasets. We used several real-world datasets from the TUDataset [30] and the *EGO-Nets* datasets [15] for our experiments. See Appendix B and F for an overview of the datasets, as well as additional synthetic datasets and corresponding results. We selected these datasets as they cover a wide range of applications, consisting of both molecule datasets and graphs derived from social networks. See Appendix C for the number of Weisfeiler-Leman iterations needed to reach the stable coloring for each dataset.

6.2 Results

In the following, we present the classification accuracies and running times of the different kernels. We investigate the parameter selection for our algorithm and discuss the application of our approach for approximating the graph edit distance.

Q1: Classification Accuracy. Table 1 shows the classification accuracy of the different kernels. While on some datasets our new approaches do not outcompete all state-of-the-art methods, they are more accurate in most cases, in some cases even with a large margin to the second-placed (for example on *KKI*, *EGO-1* or *EGO-4*). While RWL* is better than our approaches on some datasets, the running time of this method is much higher, cf. Q3. WLOA also produces very good results on many datasets, but cannot compete on the *EGO-Nets* and synthetic datasets (see Appendix F). For



(a) EGO-1 **(b)** KKI **(c)** PTC_FM **(d)** IMDB-BINARY

Figure 3: Number of times a parameter combination of GWL and GWLOA was selected from $k \in \{2, 4, 8, 16\}$ and $h \in \{0, \ldots, 10\}$ based on the accuracy achieved on the test set.

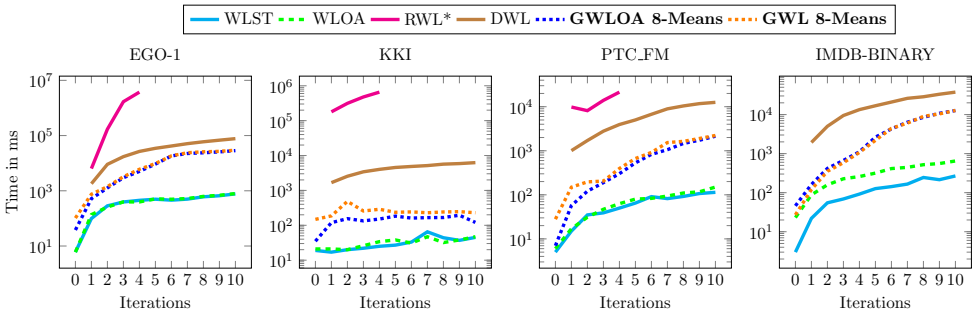


Figure 4: Running time in milliseconds for computing the feature vectors for all graphs of a dataset using the different methods. Note that RWL* uses multi-threading, while the other methods do not. Missing values for RWL* and DWL in the larger datasets are due to timeout.

molecular graphs (*PTC_FM*, *NCI*) we see no significant improvements, which can be explained by their small degree and sensitivity of molecular properties to small changes. Overall, our method provides the highest accuracy on 9 of 12 datasets and is close to the best accuracy for the others.

Q2: Parameter Selection. For GWL and GWLOA two parameters have to be chosen: The number of iterations h and the number k of clusters in k-means. We investigate which choices lead to the best classification accuracy. Figure 3 shows the number of times, a specific parameter combination was selected as it provided the best accuracy for the test set. Here, we only show the parameter selection for some of the datasets. The results for the other datasets, as well as the parameter selection for WLST and WLOA, can be found in Appendix D. We can see that for GWL and most datasets the best k is in $\{2, 4, 8\}$ and on those datasets classification accuracy of GWL exceeds that of WLST. On datasets on which GWL performed worse than WLST, the best choice for parameters is not clear and it seems like a larger k might be beneficial for improving the classification accuracy. Similar tendencies can be observed for GWLOA.

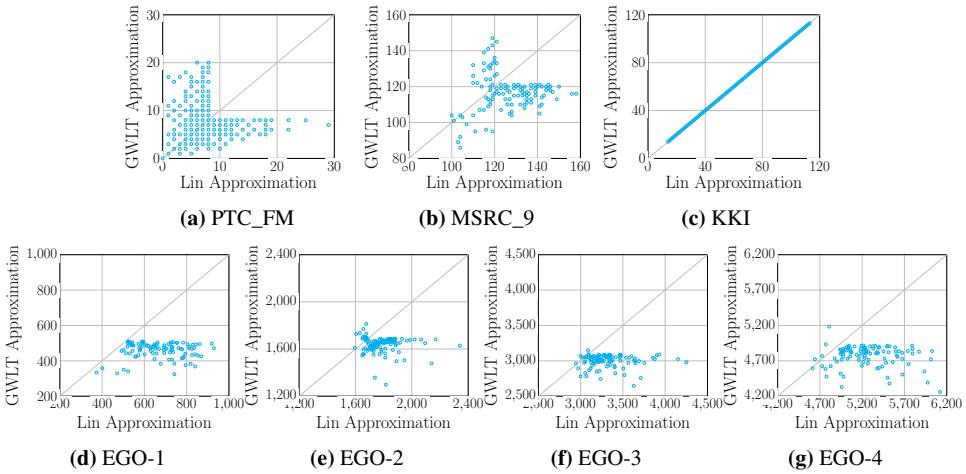
Q3: Running Time. Figure 4 shows the time needed for computing the feature vectors using the different kernels (for results on the other datasets and the influence of the parameter k on running time see Appendix E and G). RWL* and DWL are much slower than the other kernels, while only RWL* leads to minor improvements in classification accuracy on few datasets. While our approach is only slightly slower than WLST/WLOA, it yields great improvements on the classification accuracy on most datasets, cf. Q1.

Q4: Learning with Approximated Graph Edit Distance. Table 2 compares the 1-nn classification accuracy of our approach when approximating the graph edit distance to the original [27] and another state-of-the-art method based on bipartite graph matching [28]. Our approach clearly outcompetes both methods on all datasets.

We investigate the approximation quality for similar graphs (which are important for k-nn classification) further by comparing the approximation of Lin [27] to our method GWLT directly, see Figure 5.

Table 2: Average classification accuracy and standard deviation (highest accuracies marked in **bold**).

Method	PTC_FM	MSRC_9	KKI	EGO-1	EGO-2	EGO-3	EGO-4
BGM	60.14 ±1.50	72.13 ±1.28	43.89 ±1.27	44.75 ±1.05	42.05 ±1.25	out of time	out of time
Lin	62.38 ±1.08	81.36 ±0.64	**55.18** ±2.44	40.40 ±1.17	31.65 ±1.07	26.60 ±0.94	36.55 ±1.72
GWLT	**63.19** ±0.11	**85.97** ±0.59	**55.18** ±2.44	**56.20** ±1.42	**47.90** ±1.28	**36.40** ±1.04	**47.90** ±1.32



(a) PTC_FM **(b)** MSRC_9 **(c)** KKI

(d) EGO-1 **(e)** EGO-2 **(f)** EGO-3 **(g)** EGO-4

Figure 5: Approximation quality of GWLT compared to Lin [27]. Marks below the gray diagonal indicate that GWLT had a better approximation, while marks above indicate the same for Lin.

Only graph pairs for which at least one of the methods returned a distance below a cutoff threshold are depicted to emphasize the important case of highly similar graphs. The results clearly show that GWLT provides more accurate approximations, in particular, for the *EGO-Nets*. This finding and the accuracy results discussed in Q1 indicate that on these datasets methods benefit from the more fine-grained vertex similarity provided by GWL.

7 Conclusions

We proposed a general framework for iterative vertex refinement generalizing the popular Weisfeiler-Leman algorithm and discussed connections to other vertex refinement strategies. Based on this, we proposed two new graph kernels and showed that they outperform the original Weisfeiler-Leman subtree kernel and similar state-of-the-art approaches in terms of classification accuracy in almost all cases, while keeping the running time much lower than comparable methods. We also investigated the application of our method to approximating the graph edit distance, where we again outperformed the state-of-the-art methods.

In further research, other renep functions can be explored, for example, by using different clustering strategies or developing new concepts for inexact neighborhood comparison. Moreover, we will systematically relate our approach to graph neural networks and investigate whether similar ideas can be incorporated into their neighborhood aggregation step and to what extent common architectures and optimization methods are capable of learning certain renep functions.

Acknowledgements

This work has been supported by the Vienna Science and Technology Fund (WWTF) through project VRG19-009.

References

[1] Brendan D. McKay and Adolfo Piperno. Practical graph isomorphism, II. *J. Symb. Comput.*, 60:94–112, 2014. doi: 10.1016/j.jsc.2013.09.003. 1

[2] Nino Shervashidze, Pascal Schweitzer, Erik Jan van Leeuwen, Kurt Mehlhorn, and Karsten M. Borgwardt. Weisfeiler-Lehman graph kernels. *J. Mach. Learn. Res.*, 12:2539–2561, 2011. 1, 2, 4, 6, 7

[3] Matteo Togninalli, M. Elisabetta Ghisu, Felipe Llinares-López, Bastian Rieck, and Karsten M. Borgwardt. Wasserstein Weisfeiler-Lehman graph kernels. In *Advances in Neural Information Processing Systems, NeurIPS*, pages 6436–6446, 2019. 1, 3

[4] Dai Hai Nguyen, Canh Hao Nguyen, and Hiroshi Mamitsuka. Learning subtree pattern importance for Weisfeiler-Lehman based graph kernels. *Mach. Learn.*, 110(7):1585–1607, 2021. 7

[5] Nils M. Kriege, Pierre-Louis Giscard, and Richard C. Wilson. On valid optimal assignment kernels and applications to graph classification. In *Advances in Neural Information Processing Systems, NIPS*, pages 1615–1623, 2016. 1, 2, 3, 7

[6] Christopher Morris, Matthias Fey, and Nils M. Kriege. The power of the Weisfeiler-Leman algorithm for machine learning with graphs. In *International Joint Conference on Artificial Intelligence, IJCAI*, pages 4543–4550. ijcai.org, 2021. doi: 10.24963/ijcai.2021/618. 1

[7] Christopher Morris, Yaron Lipman, Haggai Maron, Bastian Rieck, Nils M. Kriege, Martin Grohe, Matthias Fey, and Karsten M. Borgwardt. Weisfeiler and Leman go machine learning: The story so far. *CoRR*, abs/2112.09992, 2021. 1, 3

[8] Christopher Morris, Martin Ritzert, Matthias Fey, William L. Hamilton, Jan Eric Lenssen, Gaurav Rattan, and Martin Grohe. Weisfeiler and Leman go neural: Higher-order graph neural networks. In *AAAI Conference on Artificial Intelligence, AAAI*, pages 4602–4609. AAAI Press, 2019. doi: 10.1609/aaai.v33i01.33014602. 1

[9] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks? In *International Conference on Learning Representations, ICLR*. OpenReview.net, 2019. 1, 3

[10] Giannis Nikolentzos and Michalis Vazirgiannis. Message passing graph kernels. *CoRR*, abs/1808.02510, 2018. 2

[11] Nils M. Kriege, Fredrik D. Johansson, and Christopher Morris. A survey on graph kernels. *Applied Network Science*, 5(1):6, 2020. doi: 10.1007/s41109-019-0195-3. 2

[12] Karsten M. Borgwardt, M. Elisabetta Ghisu, Felipe Llinares-López, Leslie O'Bray, and Bastian Rieck. Graph kernels: State-of-the-art and future challenges. *Found. Trends Mach. Learn.*, 13 (5-6), 2020. doi: 10.1561/2200000076. 2

[13] Pinar Yanardag and S. V. N. Vishwanathan. Deep graph kernels. In *International Conference on Knowledge Discovery and Data Mining, SIGKDD*, pages 1365–1374. ACM, 2015. doi: 10.1145/2783258.2783417. 2, 7

[14] Annamalai Narayanan, Mahinthan Chandramohan, Lihui Chen, Yang Liu, and Santhoshkumar Saminathan. subgraph2vec: Learning distributed representations of rooted sub-graphs from large graphs. *CoRR*, abs/1606.08928, 2016. 2

[15] Till Hendrik Schulz, Tamás Horváth, Pascal Welke, and Stefan Wrobel. A generalized Weisfeiler-Lehman graph kernel. *Mach Learn*, 2022. 2, 7, 8, 13

[16] Zhongxi Fang, Jianming Huang, Xun Su, and Hiroyuki Kasai. Wasserstein graph distance based on l_1-approximated tree edit distance between Weisfeiler-Lehman subtrees. *CoRR*, abs/2207.04216, 2022. doi: 10.48550/arXiv.2207.04216. 2

[17] Shohei Hido and Hisashi Kashima. A linear-time graph kernel. In Wei Wang, Hillol Kargupta, Sanjay Ranka, Philip S. Yu, and Xindong Wu, editors, *International Conference on Data Mining, ICDM*, pages 179–188. IEEE Computer Society, 2009. doi: 10.1109/ICDM.2009.30. 2

[18] Nino Shervashidze. *Scalable graph kernels*. PhD thesis, Eberhard Karl University of Tübingen, 2012. 2

[19] Marion Neumann, Roman Garnett, Christian Bauckhage, and Kristian Kersting. Propagation kernels: efficient graph kernels from propagated information. *Mach. Learn.*, 102(2):209–245, 2016. doi: 10.1007/s10994-015-5517-9. 2

[20] Christopher Morris, Nils M. Kriege, Kristian Kersting, and Petra Mutzel. Faster kernels for graphs with continuous attributes via hashing. In *International Conference on Data Mining, ICDM*, pages 1095–1100. IEEE Computer Society, 2016. doi: 10.1109/ICDM.2016.0142. 2

[21] Martin Grohe, Kristian Kersting, Martin Mladenov, and Pascal Schweitzer. Color Refinement and Its Applications. In *An Introduction to Lifted Probabilistic Inference*. The MIT Press, 08 2021. ISBN 9780262365598. 3

[22] Boris Weisfeiler and Adrian Leman. A reduction of a graph to canonical form and an algebra arising during this reduction. *Nauchno-Technicheskaya Informatsia*, 2(9):12–16, 1968. 3

[23] Sandra Kiefer and Brendan D. McKay. The iteration number of colour refinement. In *International Colloquium on Automata, Languages, and Programming, ICALP*, volume 168 of *LIPIcs*, pages 73:1–73:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. doi: 10.4230/LIPIcs.ICALP.2020.73. 3, 5

[24] Christoph Berkholz, Paul S. Bonsma, and Martin Grohe. Tight lower and upper bounds for the complexity of canonical colour refinement. *Theory Comput. Syst.*, 60(4):581–614, 2017. doi: 10.1007/s00224-016-9686-0. 5, 6

[25] Robert Paige and Robert Endre Tarjan. Three partition refinement algorithms. *SIAM J. Comput.*, 16(6):973–989, 1987. doi: 10.1137/0216062.

[26] Tommi A. Junttila and Petteri Kaski. Engineering an efficient canonical labeling tool for large and sparse graphs. In *Workshop on Algorithm Engineering and Experiments, ALENEX*. SIAM, 2007. doi: 10.1137/1.9781611972870.13. 6

[27] Nils M. Kriege, Pierre-Louis Giscard, Franka Bause, and Richard C. Wilson. Computing optimal assignments in linear time for approximate graph matching. In *International Conference on Data Mining, ICDM*, pages 349–358, 2019. 7, 9, 10, 17

[28] Kaspar Riesen and Horst Bunke. Approximate graph edit distance computation by means of bipartite graph matching. *Image Vision Comput.*, 27(7):950–959, 2009. 7, 9, 17

[29] Chih-Chung Chang and Chih-Jen Lin. LIBSVM: A library for support vector machines. *ACM Transactions on Intelligent Systems and Technology*, 2:27:1–27:27, 2011. Software available at `http://www.csie.ntu.edu.tw/~cjlin/libsvm`. 8

[30] Christopher Morris, Nils M. Kriege, Franka Bause, Kristian Kersting, Petra Mutzel, and Marion Neumann. TUDataset: A collection of benchmark datasets for learning with graphs. In *ICML 2020 Workshop on Graph Representation Learning and Beyond, GRL+*, 2020. URL `http://www.graphlearning.io`. 8, 13

[31] Rong-En Fan, Kai-Wei Chang, Cho-Jui Hsieh, Xiang-Rui Wang, and Chih-Jen Lin. LIBLINEAR: A library for large linear classification. *Journal of Machine Learning Research*, 9: 1871–1874, 2008. 17

[32] Zhiping Zeng, Anthony K. H. Tung, Jianyong Wang, Jianhua Feng, and Lizhu Zhou. Comparing stars: On approximating graph edit distance. *Proc. VLDB Endow.*, 2(1):25–36, 2009. 17

Table 3: Datasets with discrete vertex and edge labels and their statistics [30]. The *EGO-Nets* datasets [15] are unlabeled.

Name	|Graphs|	|Classes|	avg $\|V\|$	avg $\|E\|$	$\|L_V\|$	$\|L_E\|$
KKI	83	2	26.96	48.42	190	–
PTC_FM	349	2	14.11	14.48	18	4
COLLAB	5000	3	74.49	2457.78	–	–
DD	1178	2	284.32	715.66	82	–
IMDB-BINARY	1000	2	19.77	96.53	–	–
MSRC_9	221	2	40.58	97.94	10	–
NCI1	4110	2	29.87	32.30	37	–
REDDIT-BINARY	2000	2	429.63	497.75	–	–
EGO-1	200	4	138.97	593.53	–	–
EGO-2	200	4	178.55	1444.86	–	–
EGO-3	200	4	220.01	2613.49	–	–
EGO-4	200	4	259.78	4135.80	–	–

A Pseudocode

Algorithm 1 shows the procedure of gradual Weisfeiler-Leman refinement. We start with an initial coloring (either uniform or based on the vertex labels) and then iteratively refine these colors using a renep function, h times.

Algorithm 1 Gradual Weisfeiler-Leman Refinement.

procedure GWL(G, h) ▷ h is number of iterations
  for all $v \in V(G)$ **do** ▷ Initial coloring
    $c_0(v) \leftarrow \mu(v)$
  initialize T_0 as described in Section 3
  for $i \leftarrow 1, i \leq h, i \leftarrow i+1$ **do**
    $\mathcal{T}_i \leftarrow f(G, \mathcal{T}_{i-1})$ ▷ Compute new colors
    for all $v \in V(G)$ **do** ▷ Assign new colors
      $c_i(v) \leftarrow \pi_{\mathcal{T}_i}(v)$

B Datasets

We used several real-world datasets from the TUDataset [30], the *EGO-Nets* datasets [15], as well as synthetic datasets for our experiments. See Table 3 for an overview of the real-world datasets. We selected these datasets as they cover a wide range of applications, consisting of both molecule datasets and graphs derived from social networks.

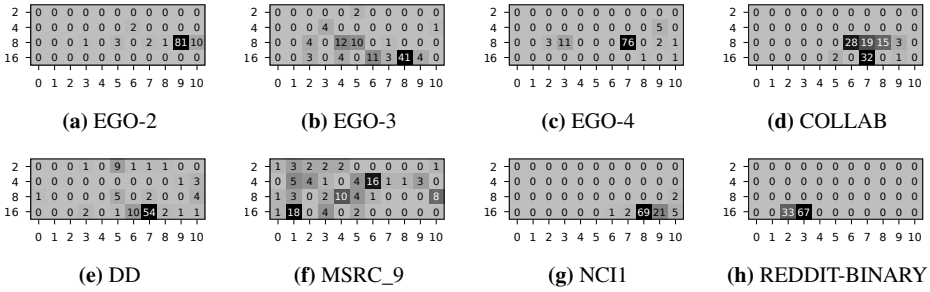
The synthetic datasets were generated using the block graph generation method [15]. We generated 9 synthetic datasets with two classes and 200 graphs in each class. For each dataset we first generated two seed graphs (one per class) with 16 vertices, that both are constructed from a tree by appending a single edge, so that their sets of vertex degrees are equal. For the dataset graphs each vertex of the seed graph was replaced by 8 vertices. Vertices generated from the same seed vertex, as well as from adjacent vertices are connected with probability p. m noise edges are then added randomly. We investigated the two cases $p = 1.0$ and $m \in \{0, 10, 20, 50, 100\}$, and $p \in \{1.0, 0.8, 0.6, 0.4, 0.2\}$ and $m = 0$. We denote the datasets by *S_p_m*.

C Number of Weisfeiler-Leman Iterations

We investigate the number of WL iterations needed to reach the stable coloring in the various datasets. On the datasets with $-$ entries (see Table 4) our algorithm, that checked whether the stable coloring is reached, did not finish in a reasonable time due to the size of the datasets/graphs. It can be seen

Table 4: Number of iterations needed to reach the stable coloring.

Dataset	*KKI*	*PTC_FM*	*COLLAB*	*DD*	*IMDB-B*	*MSRC_9*
WL	3	13	–	–	3	3
Dataset	*NCI1*	*REDDIT-B*	*EGO-1*	*EGO-2*	*EGO-3*	*EGO-4*
WL	39	–	5	4	4	5



(a) EGO-2 **(b)** EGO-3 **(c)** EGO-4 **(d)** COLLAB

(e) DD **(f)** MSRC_9 **(g)** NCI1 **(h)** REDDIT-BINARY

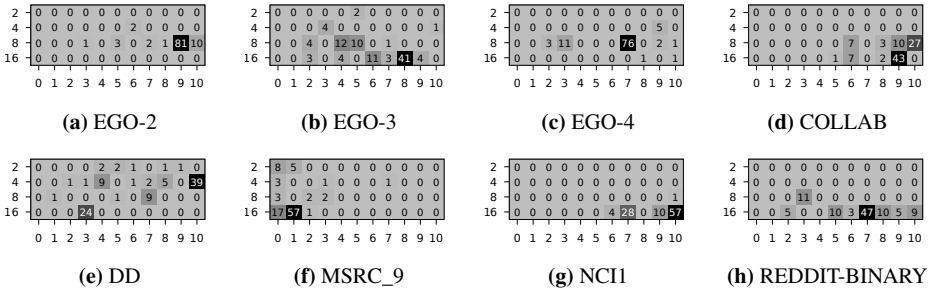
Figure 6: With $k \in \{2, 4, 8, 16\}$ and $h \in \{0, \dots, 10\}$, we show the number of times a specific parameter combination for GWL was selected as it provided the best accuracy for the test set.

that on most datasets the number of iterations needed to reach the stable coloring is very low. This means that after a few iterations we do not gain any new information when using for example the traditional WLST kernel.

D Parameter Selection - Further Results

Figures 6 and 7 show the parameter selection for the remaining datasets for GWL and GWLOA. In most datasets the choice is restricted to two or three values. For some of the datasets, the best choice seems to include $k = 16$. This might indicate that a larger k could be beneficial for increasing the accuracy.

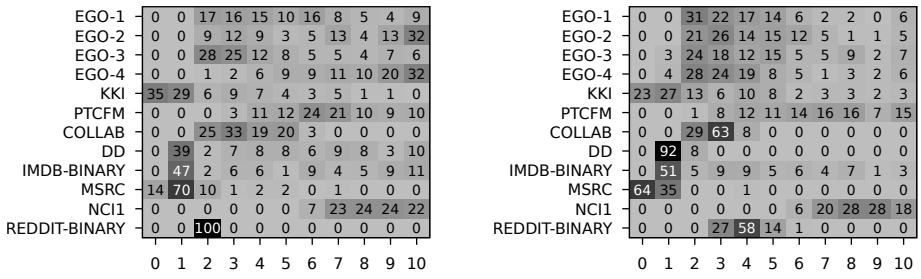
Figure 8 shows the parameter selection for WLST and WLOA. There is only one parameter, the number of WL iterations, for both kernels. We can see that indeed on most datasets, only few iterations are needed to gain the best possible accuracy. On datasets such as *EGO-2*, *EGO-4* or *NCI1*, however, this is not the case. For *NCI1* we can assume that we still gain information through more iterations, since we have not yet reached the stable coloring. For the *EGO*-datasets, this is surprising, since the stable coloring is reached after 4 (5) iterations. On the other hand, the classification accuracy reached is still not good.



(a) EGO-2 **(b)** EGO-3 **(c)** EGO-4 **(d)** COLLAB

(e) DD **(f)** MSRC_9 **(g)** NCI1 **(h)** REDDIT-BINARY

Figure 7: With $k \in \{2, 4, 8, 16\}$ and $h \in \{0, \dots, 10\}$, we show the number of times a specific parameter combination for GWLOA was selected as it provided the best accuracy for the test set.



(a) WLST **(b)** WLOA

Figure 8: With $h \in \{0, \ldots, 10\}$, we show the number of times a specific parameter combination for WLST and WLOA was selected as it provided the best accuracy for the test set.

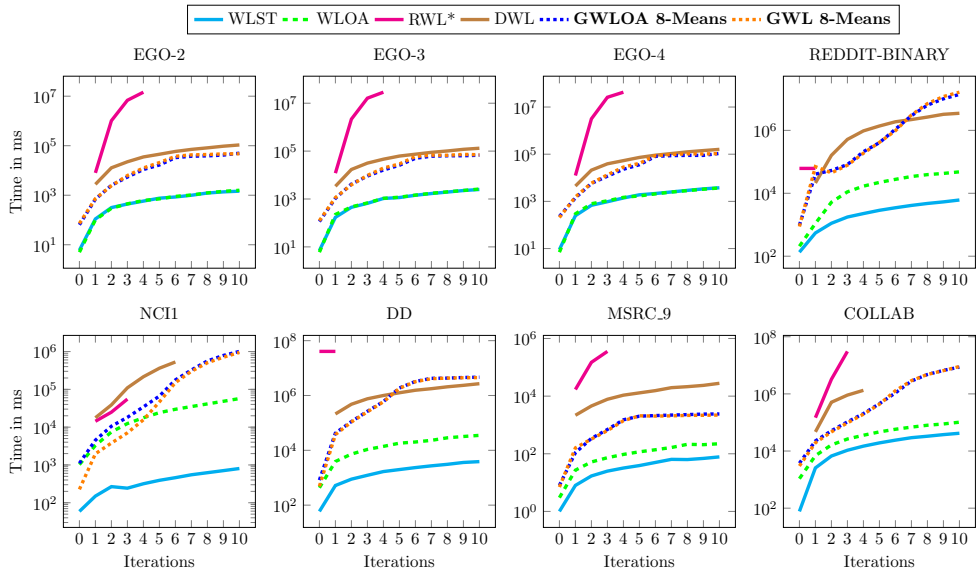


Figure 9: Running time in milliseconds for computing the feature vectors using the different methods. Note that RWL* uses multi-threading, while the other methods do not. Missing values for RWL* and DWL in the larger datasets are due to timeout.

E Running Time - Further Results

Figure 9 shows the running time results for the remaining datasets. While the runtime of our approach exceeds that of DWL on some of the larger datasets, it enhances the classification accuracy a lot.

F Results on Synthetic Datasets

Table 5 shows the classification accuracy of the different methods on the synthetic datasets (generated as described in Appendix B), with the best accuracy for each dataset being marked in bold. We can see, while all kernels can perfectly learn on the datasets without noise, neither WLST, WLOA nor DWL can manage the noise included in the other datasets, having worse accuracy with increasing noise. While the decrease in accuracy with decreasing the edge probability is slightly worse than that of RWL*, our approach has a much lower running time.

Table 5: Average classification accuracy and standard deviation on the synthetic datasets.

Kernel	S_1_0	S_1_10	S_1_20	S_1_50	S_1_100	$S\_0.8\_0$	$S\_0.6\_0$	$S\_0.4\_0$	$S\_0.2\_0$
WLST	**100.00**±0.00	98.68 ±0.34	61.93 ±1.08	54.55 ±0.68	49.78 ±1.18	50.65 ±1.57	48.10 ±1.10	51.98 ±1.40	42.65 ±1.80
DWL	**100.00**±0.00	98.70 ±0.31	62.10 ±0.98	43.83 ±1.66	51.40 ±0.94	49.80 ±2.01	46.88 ±1.89	49.05 ±1.98	42.85 ±1.98
RWL*	**100.00**±0.00	**100.00**±0.00	**100.00**±0.00	out of time	**100.00**±0.00	**100.00**±0.00	**99.35** ±0.17	**81.93** ±1.00	**56.33** ±2.48
WLOA	**100.00**±0.00	97.65 ±0.44	60.85 ±1.65	47.50 ±1.83	50.23 ±1.24	49.45 ±1.47	48.08 ±1.92	43.23 ±1.21	50.53 ±1.92
GWL	**100.00**±0.00	**100.00**±0.00	**100.00**±0.00	**100.00**±0.00	**100.00**±0.00	**100.00**±0.00	92.20 ±0.97	72.53 ±1.78	53.58 ±1.71
GWLOA	**100.00**±0.00	**100.00**±0.00	**100.00**±0.00	**100.00**±0.00	**100.00**±0.00	**100.00**±0.00	94.95 ±0.59	72.03 ±1.66	50.90 ±2.63

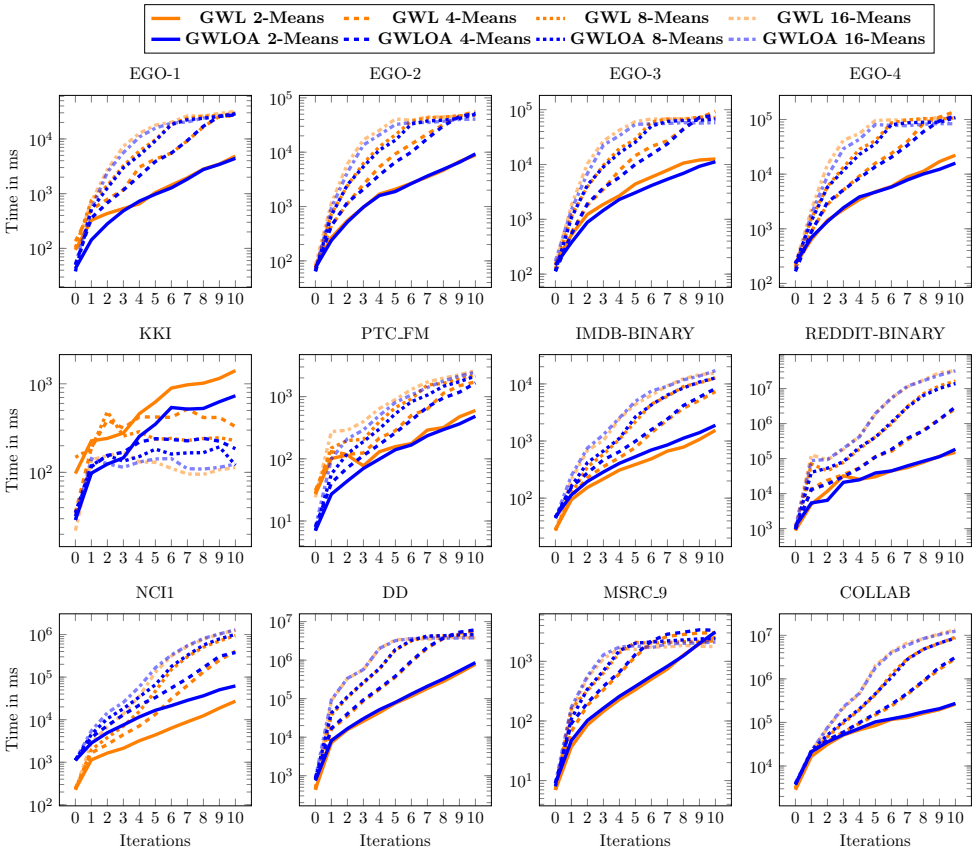


Figure 10: Running time in milliseconds for computing the feature vectors using the different values for parameter k on our newly proposed methods.

G Influence of Parameter k on Running Time

We investigate which effect the choice of k has on the running time. Figure 10 shows the time needed for computing the feature vectors using our kernels with $k \in \{2, 4, 8, 16\}$. The difference in running time between GWL and GWLOA is only marginal on most datasets, only on *KKI* and *NCI1* a larger difference can be seen. As expected, the running time of both kernels increases with increasing k. Interestingly, for larger k, the running time does not increase much anymore, after a certain number of iterations, this might be because the stable coloring was reached by then.

H Results on Larger Datasets

GWL allows to generate explicit sparse feature vectors just as WLST. Therefore, these kernels can be used with a linear SVM, which is more efficient than a kernel SVM and makes the application to larger datasets feasible. We performed additional experiments with these kernels on larger synthetic

Table 6: Parameters used for generating the larger datasets.

Dataset	p	m	|Graphs|	b	r
L1	1	200	10000	25	10
L2	1	100	20000	25	10
L3	1	200	20000	10	15
L4	1	400	20000	25	10

Table 7: Results on larger datasets with running time in seconds and $|f|$ being the number of features (for WLST we only give the order of magnitude).

Kernel	*L1*			*L2*			*L3*			*L4*		
	time	acc	$\|f\|$	time	acc	$\|f\|$	time	acc	$\|f\|$	time	acc	$\|f\|$
WLST	25	50.92	10^6	59	84.57	10^6	45	49.91	10^6	124	49.65	10^6
GWL	208	99.98	61	237	100.0	15	103	100.0	7	137	99.98	7

datasets using the linear SVM implementation LIBLINEAR [31]. The datasets were generated using the same method as described in Appendix B, but with different values for the number of vertices in the seed graphs, b, and number of vertices, each vertex in the seed graph is replaced by, r (see Table 6 for the values of the parameters). The experimental setup used in these experiments was as follows: We split the dataset randomly into a training, validation and test set and chose the parameter $C \in \{0.1, 1, 10\}$. We used $h \in \{1, \dots, 5\}$ for both kernels and set $k = 2$ for all datasets.

In Table 7 we report the running time and the number of features f obtained for the choice of h that gave the best classification accuracy (for GWL this choice was $h = 2$ for *L4* and *L5*, $h = 3$ for *L2* and $h = 5$ for *L1*, for WLST it was $h = 2$ for all datasets except *L4*, on which it was $h = 5$). We noticed that running the SVM with the feature vectors generated by WLST took much more time than GWL, which can be explained by the number of features each algorithm produced: For GWL the number is restricted by the choice of parameters, but for WLST the number of features is only restricted by the number of vertices in the dataset. In WLST the number of features exceeded 10^6 for $h \geq 2$. For a small increase in running time in generating the feature vectors, GWL provides not only a much better classification accuracy than WLST, it also generates less features, but more meaningful ones.

I Approximating the Graph Edit Distance using Optimal Assignments

The graph edit distance (GED), a commonly used distance measure for graphs, is defined as the cost of transforming one graph into the other using edit operations, i.e. deleting or inserting an isolated vertex or an edge, or relabeling any of the two. An edit path between G and H is a sequence $(e_1, e_2, \dots, e_k)$ of edit operations that transforms G into H. This means, that applying all operations in the edit path to G, yields a graph G' that is isomorphic to H.

Edit operations have non-negative costs assigned to them by a cost function c and the GED of two graphs is defined as the cost of a cheapest edit path between them:

$$\mathrm{GED}(G, H) = \min \left\{ \sum_{i=1}^{k} c(e_i) \mid (e_1, \dots, e_k) \in \Upsilon(G, H) \right\},$$

where $\Upsilon(G, H)$ denotes the set of all possible edit paths from G to H.

Since computing the GED is NP-hard [32], it is often approximated; often an optimal assignment between the vertices of the graphs is computed and a (suboptimal) edit path is derived from this assignment [28]. Figure 11 shows two graphs, an assignment between their vertices indicated by matching numbers, and a corresponding edit path.

If the cost function is a tree metric, such an assignment can be computed in linear time [27] (as opposed to cubic runtime for arbitrary cost functions). The color hierarchy produced by for example GWLT can be interpreted as such a tree metric, which means it can be used to find an optimal

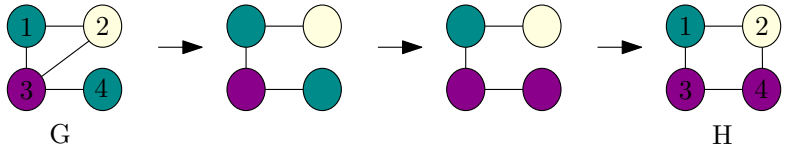


Figure 11: Two graphs G and H with an assignment between their vertices (vertices with matching numbers are assigned to each other) and an edit path derived from this assignment.

assignment between the vertices and in turn approximate the graph edit distance. Given a tree T, representing a tree metric, an optimal assignment between two sets A and B can be computed as follows: Let ϕ be a function that associates all elements with their node in the tree (for GWLT this means, that we associate each vertex with its "newest" color). Then deconstruct T fully (until there are no leaves left) by repeating these steps:

1. Pick a random leaf l.
2. Assign as many elements of A associated with l to elements of B associated with l as possible.
3. Re-associate remaining elements to the parent of l.
4. Delete l from T.

The resulting assignment is optimal and can be used to derive an edit path as above.

A.5 On the Two Sides of Redundancy in Graph Neural Networks

Authors Franka Bause, Samir Moustafa, Johannes Langguth, Wilfried N. Gansterer, and Nils M. Kriege

Publication Venue Published in *Machine Learning and Knowledge Discovery in Databases. Research Track*, LNCS 14945, pages 300–318 (2024) by Springer Nature.

Acknowledgment Reproduced with permission from Springer Nature.

DOI/URL https://doi.org/10.1007/978-3-031-70365-2_22

Extended Version https://arxiv.org/abs/2310.04190

Authors Contributions

Franka Bause made significant contributions to the conception of neighborhood trees and their efficient generation (first proposed in [BPK24], see Appendix A.3), developed the methods for redundancy removal presented in Sections 4.1 and 4.2 with NK, developed Theorem 1 and the theoretical analysis, investigated the connections to related work regarding computational complexity and expressivity in Section 4.5, implemented the neighborhood trees, merge DAGs, and algorithmic components of the implementation, developed the DAG-MLP architecture with NK and SM, and drafted, wrote, and revised the manuscript with input from the other authors.

Samir Moustafa developed the DAG-MLP architecture with NK and FB, implemented DAG-MLP, conducted the experimental evaluation including environment configuration and learning pipeline integration, and wrote parts of the corresponding sections in the manuscript.

Johannes Langguth gave valuable feedback and helped to revise the manuscript.

Wilfried N. Gansterer supervised the project, gave valuable feedback, and helped to revise the manuscript.

Nils M. Kriege initiated and supervised the project, developed the methods for redundancy removal presented in Sections 4.1 and 4.2 with FB, developed the DAG-MLP architecture with FB and SM, gave valuable feedback, and helped to draft and revise the manuscript.

Reference in Thesis [BML+24]

Abstract

Message passing neural networks iteratively generate node embeddings by aggregating information from neighboring nodes. With increasing depth, information from more distant nodes is included. However, node embeddings may be unable to represent the growing node neighborhoods accurately and the influence of distant nodes may vanish, a problem referred to as oversquashing. Information redundancy in message passing, i.e., the repetitive exchange and encoding of identical information amplifies oversquashing. We develop a novel aggregation scheme based on neighborhood trees, which allows for controlling redundancy by pruning redundant branches of unfolding trees underlying standard message passing. While the regular structure of unfolding trees allows the reuse of intermediate results in a straightforward way, the use of neighborhood trees poses computational challenges. We propose compact representations of neighborhood trees and merge them, exploiting computational redundancy by identifying isomorphic subtrees. From this, node and graph embeddings are computed via a neural architecture inspired by tree canonization techniques. Our method is less susceptible to oversquashing than traditional message passing neural networks and can improve the accuracy on widely used benchmark datasets.

On the Two Sides of Redundancy in Graph Neural Networks

Franka Bause[1,2(✉)], Samir Moustafa[1,2], Johannes Langguth[4], Wilfried N. Gansterer[1], and Nils M. Kriege[1,3]

[1] Faculty of Computer Science, University of Vienna, Vienna, Austria
{franka.bause,samir.moustafa,wilfried.gansterer,nils.kriege}@univie.ac.at
[2] UniVie Doctoral School Computer Science, University of Vienna, Vienna, Austria
[3] Research Network Data Science, University of Vienna, Vienna, Austria
[4] Simula Research Laboratory, Oslo, Norway
langguth@simula.no

Abstract. Message passing neural networks iteratively generate node embeddings by aggregating information from neighboring nodes. With increasing depth, information from more distant nodes is included. However, node embeddings may be unable to represent the growing node neighborhoods accurately and the influence of distant nodes may vanish, a problem referred to as oversquashing. Information redundancy in message passing, i.e., the repetitive exchange and encoding of identical information amplifies oversquashing. We develop a novel aggregation scheme based on neighborhood trees, which allows for controlling redundancy by pruning redundant branches of unfolding trees underlying standard message passing. While the regular structure of unfolding trees allows the reuse of intermediate results in a straightforward way, the use of neighborhood trees poses computational challenges. We propose compact representations of neighborhood trees and merge them, exploiting computational redundancy by identifying isomorphic subtrees. From this, node and graph embeddings are computed via a neural architecture inspired by tree canonization techniques. Our method is less susceptible to oversquashing than traditional message passing neural networks and can improve the accuracy on widely used benchmark datasets.

Keywords: Graph neural networks · Oversquashing · Non-redundant message passing

1 Introduction

Graph neural networks (GNNs) emerged as the dominant approach for machine learning on graph data, with the class of message passing neural networks (MPNNs) [13] being widely used. These networks update node embeddings layer wise by combining the current embedding of a node with those of its neighbors, involving learnable parameters. Suitable neural architectures, which admit a parametrization such that each layer represents an injective function uniquely encoding the input, have the same expressive power as the Weisfeiler-Leman

A. Bifet et al. (Eds.): ECML PKDD 2024, LNAI 14946, pp. 371–388, 2024.
https://doi.org/10.1007/978-3-031-70365-2_22

algorithm [36]. The Weisfeiler-Leman algorithm distinguishes two nodes if and only if the unfolding trees representing their neighborhoods are non-isomorphic. These unfolding trees correspond to the computational trees of MPNNs [16,31]. Hence, nodes with isomorphic unfolding trees will obtain the same embedding, while for nodes with non-isomorphic unfolding trees, there exist parameters such that their embeddings differ. This implies that deeper unfolding trees lead to more expressive methods. Despite this theoretical connection, shallow MPNNs are often favored in practice. Challenges arise from the observed convergence of node embeddings for deep architectures, referred to as *oversmoothing* [21,22], and the issue of *oversquashing* [5], where the node neighborhood grows exponentially with the number of layers, and therefore, cannot be supposed to be accurately represented by a fixed-sized embedding. Recently, oversquashing has been investigated by analyzing the sensitivity of node embeddings to the initial features of distant nodes, relating the phenomenon to the *graph curvature* [34], the *effective resistance* [8] and the *commute time* [14]. On this basis several graph rewiring strategies have been proposed to mitigate oversquashing [8,34].

We address the problem of oversquashing by modifying the message passing scheme for eliminating the encoding of repeated information. For example, in an undirected graph, when a node sends information to its neighbour, future messages sent back via the same edge will contain the exact information previously sent, leading to redundancy. In the context of walk-based graph learning this problem is well-known and referred to as *tottering* [23]. Recent work by Chen et al. [9] established a first result formalizing the relation between redundancy and oversquashing by sensitivity analysis. Several recent GNNs replace the walk-based aggregation by mechanisms based on simple or shortest paths reporting promising results [2,17,27]. PathNNs [27] and RFGNN [17] are closely related approaches, defining path-based trees for nodes and employing custom aggregation schemes. However, these methods suffer from high computational costs compared to standard MPNNs and often have an exponential time complexity. The crucial advantage of MPNNs is the regular structure of aggregations applied through all layers, while reducing information redundancy leads to a less regular structure, rendering it challenging to exploit computational redundancy.

Our Contribution. We systematically explore the issue of information redundancy within MPNNs and introduce principled techniques to eliminate superfluous messages. Our investigation is based on the implicit tree representation used by both MPNNs and the Weisfeiler-Leman algorithm. We first develop a neural tree canonization approach that systematically processes trees in a bottom-up fashion and extend it to directed acyclic graphs (DAGs). To exploit computational redundancy, we merge trees representing node neighborhoods into a DAG, identifying isomorphic subtrees. Our approach, termed DAG-MLP, recovers the computational graph of MPNNs for unfolding trees, while avoiding redundant computations in the presence of symmetries. We employ the canonization technique on *neighborhood trees*, which are derived from unfolding trees by eliminating redundant nodes. We show that neighborhood trees allow distinguishing

Table 1. Time complexity of preprocessing, size of computation graph and expressivity of our method compared to related work. n: number of nodes, m: number of edges, b: maximum node degree, K: path length, h: tree height, L: number of layers, and $m_2 = 0.5 \sum_{v \in V} |N_2(v)|$.

Method	Preprocessing	Size Comp. Graph/Runtime	Expressivity
PathNet	$O(mb)$	$O(2^L(m + m_2))$	n/a
PathNN-SP	$O(nb^K)$	$O(nbK)$	incomparable
PathNN-SP+	$O(nb^K)$	$O(nbK)$	$>$ 1-WL
RFGNN	$O(n!/(n-h-1)!)$	$O(n!/(n-h-1)!)$	incomparable
DAG-MLP (0-/1-NTs)	$O(nm)$	$O(nm)$	incomparable

nodes that are indistinguishable by the Weisfeiler-Leman algorithm. The DAGs derived from neighborhood trees have size at most $O(nm)$ for input graphs with n nodes and m edges making the approach computational feasible. We formally show by sensitivity analysis that our approach reduces oversquashing. Our approach achieves high accuracy across various node and graph classification tasks.

2 Related Work

The graph isomorphism network (GIN) [36] is an MPNN that generalizes the Weisfeiler-Leman algorithm, achieving its expressive power. The limited expressivity of simple MPNNs has led to an increased interest in researching more powerful architectures, such as encoding graph structure as additional features or modifying the message passing procedure. Shortest Path Networks [2] use multiple aggregation functions for different shortest path lengths, allowing direct communication with distant nodes. While this might help mitigate oversquashing, information about the structure of the neighborhood still cannot be represented adequately and the gain in expressivity is limited. Distance Encoding GNNs [20] encode the distances of nodes to a set of target nodes. While being provably more expressive than the standard WL algorithm, the approach is limited to solving node-level tasks, as the encoding depends on a fixed set of target nodes and has not been employed for graph-level tasks. MixHop [3] concatenates results from activation functions for each neighborhood, but in contrast to Shortest Path Networks [2], the aggregation is based on normalized powers of the adjacency matrix, not shortest paths, which fails to solve the problem of redundant messages. SPAGAN [38] proposes a path-based attention mechanism, sampling shortest paths and using them as features. However, a theoretical investigation is lacking and the approach utilizes only one layer. Short-rooted random walks in [33] capture long-range dependencies, but have notable limitations due to sampling paths. The evaluation is restricted to node classification datasets and an extensive study of their expressive power is lacking. IDGNN [39] tracks the identity of the root node in unfolding trees, achieving higher expressivity than 1-WL, but failing to reduce redundant information aggregation. PathNNs [27]

Fig. 1. Graph G and its unfolding trees F_2^v for all $v \in V(G)$.

define path-based trees and a custom aggregation scheme, but overlook exploiting computational redundancy. RFGNNs [9] aim to reduce redundancy by altering the message flow to only include each node (except for the root node) at most once in each path of the computational tree. While this reduces redundancy to some extent, nodes and even the same subpaths may repeatedly occur in the computational trees. The size and running time complexity of RFGNN are very restrictive. While the term breadth-first search used in the definition of the underlying TPTs suggests linear running time, the breadth-first search has to be modified, leading to exponential running time. Additionally, redundancy in computation is not addressed resulting in a highly inefficient preprocessing and computation, limiting the method to a maximum of 3 layers in the experiments.

These architectures lack thorough investigation of their expressivity and connections to other approaches. Importantly, they do not explicitly investigate both types of redundancy in MPNNs – redundancy in the information flow and in computation. We compare the time complexity, as well as the expressivity of our method DAG-MLP and other relevant methods in Table 1 and further discuss it in Sect. 4.5.

3 Preliminaries

Graph Theory. A *graph* $G = (V, E, \mu, \nu)$ consists of a set of vertices V, a set of edges $E \subseteq V \times V$ between them, and functions $\mu\colon V \to X$ and $\nu\colon E \to X$ assigning arbitrary attributes to the vertices and edges, respectively.[1] An edge from u to v is denoted by uv, and in undirected graphs $uv = vu$. The vertices and edges of a graph G are denoted by $V(G)$ and $E(G)$, respectively. The *neighbors* (or in-neighbors) of a vertex $u \in V$ are denoted by $N(u) = \{v \mid vu \in E\}$, and the *out-neighbors* of a vertex $u \in V$ are denoted by $N_o(u) = \{v \mid uv \in E\}$. A *multigraph* is a graph, where E is a multiset, allowing multiple edges between a pair of vertices. Two graphs G and H are isomorphic, denoted by $G \simeq H$, if there exists a bijection $\phi\colon V(G) \to V(H)$, such that $\forall u, v \in V(G)\colon \mu(v) = \mu(\phi(v)) \wedge uv \in E(G) \Leftrightarrow \phi(u)\phi(v) \in E(H) \wedge \forall uv \in E(G)\colon \nu(uv) = \nu(\phi(u)\phi(v))$. We refer to ϕ as an *isomorphism* between G and H.

An *in-tree* T is a connected, directed, acyclic graph with a distinct vertex $r \in V(T)$ with no outgoing edges, referred to as *root* ($r(T)$), in which $\forall v \in V(T)\backslash r(T) : |N_o(v)| = 1$. For $v \in V(T)\backslash r(T)$ the *parent* $p(v)$ is the unique vertex $u \in N_o(v)$, and $\forall v \in V(T)$ the *children* are defined as $\text{chi}(v) = N(v)$.

[1] Edge attributes are not considered in the following for clarity of presentation, though the proposed methods can be extended to incorporate them.

(a) Graph G (b) F_2^v (c) $T_{2,0}^v$ (d) $T_{2,1}^v$

Fig. 2. Graph G and the unfolding, 0- and 1-redundant neighborhood trees of height 2 of vertex v (vertex in the upper left of G).

We refer to vertices without incoming edges as *leaves*, denoted by $l(T) = \{v \in V(T) \mid \mathrm{chi}(v) = \emptyset\}$. Conceptually an in-tree is a directed tree, in which there is a unique directed path from each vertex to the root [26]. In our paper, we only consider in-trees and will therefore refer to them simply as *trees*. In-trees are generalized by directed, acyclic graphs (DAGs). The *leaves* of a DAG D and the *children* of a vertex are defined as in trees. However, there can be multiple roots, and a vertex may have more than one parent. We refer to all vertices in D without outgoing edges as *roots*, denoted by $r(D) = \{v \in V(D) \mid N_o(v) = \emptyset\}$, and define the *parents* $p(v)$ of a vertex v as $p(v) = N_o(v)$. The height hgt of a node v is the length of the longest path from any leaf to v: $\mathrm{hgt}(v) = 0$, if $v \in l(D)$ and $\mathrm{hgt}(v) = \max_{c \in \mathrm{chi}(v)} \mathrm{hgt}(c) + 1$, otherwise. The height of a DAG D is defined as $\mathrm{hgt}(D) = \max_{v \in V(D)} \mathrm{hgt}(v)$. For clarity we refer to the vertices of a DAG as nodes to distinguish them from the graphs that are the input of a graph neural network.

Weisfeiler-Leman and Message Passing Neural Networks. The 1-dim. Weisfeiler-Leman (WL) algorithm, also known as *color refinement*, starts with vertices having a color corresponding to their label (or a uniform coloring for unlabeled vertices). In each iteration the vertex color is updated based on the multiset of colors of its neighbors according to

$$c_{\mathsf{wl}}^{(i+1)}(v) = h\left(c_{\mathsf{wl}}^{(i)}(v), \{\!\{c_{\mathsf{wl}}^{(i)}(u) \mid u \in N(v)\}\!\}\right) \quad \forall v \in V(G),$$

where h is an injective function, typically using integers to represent colors.

The color of a vertex encodes its neighborhood through a tree T, which may contain multiple representatives of each vertex. Let $\phi\colon V(T) \to V(G)$ be a mapping such that $\phi(n) = v$ if the node n in $V(T)$ represents the vertex v in $V(G)$. The *unfolding tree* F_i^v with height i of the vertex $v \in V(G)$ consists of a root n_v with $\phi(n_v) = v$ and child subtrees F_{i-1}^u for all $u \in N(v)$, where $F_0^v = (\{n_v\}, \emptyset)$. The attributes of the original graph are preserved, as illustrated in Fig. 1. The unfolding trees F_i^v and F_i^w of two vertices v and w are isomorphic if and only if $c_{\mathsf{wl}}^{(i)}(v) = c_{\mathsf{wl}}^{(i)}(w)$.

Message passing neural networks such as GIN [36] can be seen as a neural variant of the Weisfeiler-Leman algorithm. The embedding of a vertex v in layer i of GIN is defined as

$$x_i(v) = \mathrm{MLP}_i\left((1+\epsilon_i)\cdot x_{i-1}(v) + \sum_{u\in N(v)} x_{i-1}(u)\right), \tag{1}$$

where the initial features $x_0(v)$ are usually acquired by applying a multi-layer perceptron (MLP) to the vertex features.

4 Non-redundant Graph Neural Networks

We propose to restrict the information flow in message passing to regulate redundancy through the use of k-redundant neighborhood trees. We first develop a neural tree canonization technique, and obtain an MPNN via its application to unfolding trees. Subsequently, we explore computational methods on graph level, reusing information computed for subtrees and derive a customized GNN architecture. Finally, we prove that k-redundant neighborhood trees and unfolding trees are incomparable regarding their expressivity on node-level.

4.1 Removing Information Redundancy

As previously discussed, two vertices obtain the same WL color if and only if their unfolding trees are isomorphic. This concept directly carries over to message passing neural networks and their computational tree [16,31]. However, unfolding trees were mainly used as tools in expressivity analysis and as a conceptual framework for explaining mathematical properties in graph learning [19,30]. We propose a novel perspective on MPNNs through tree canonization. From this perspective, we derive a non-redundant GNN architecture based on neighborhood trees.

In their classical textbook, Aho, Hopcroft, and Ullman [4, Section 3.2] describe a linear time isomorphism test for rooted unordered trees. We give a high-level description to establish the foundation for our neural variant without focusing on the running time. The algorithm proceeds in a bottom-up manner, assigning integers $\mathsf{c}_{\mathsf{ahu}}(v)$ to each node v in the tree. The function f injectively maps a pair consisting of an integer and a multiset of integers to a new, unused integer. Initially, all leaves v are assigned integers $\mathsf{c}_{\mathsf{ahu}}(v) = f(\mu(v), \emptyset)$ based on their label $\mu(v)$. Then internal nodes are processed level-wise in a bottom-up manner, ensuring that whenever a node is processed, all its children have been considered. Hence, the algorithm computes for all nodes v of the tree

$$\mathsf{c}_{\mathsf{ahu}}(v) = f(\mu(v), \{\!\{\mathsf{c}_{\mathsf{ahu}}(u) \mid u \in \mathrm{chi}(v)\}\!\}). \tag{2}$$

This process ensures the unique representation of non-isomorphic trees, serving as the foundation of our neural tree canonization technique.

GNNs via Unfolding Tree Canonization. Combining Eq. (2) and unfolding trees, denoting the root of an unfolding tree of a vertex v of height i by n_v^i, yields

$$\begin{aligned} \mathsf{c}_{\mathsf{ahu}}(n_v^i) &= f(\mu(n_v^i), \{\!\{\mathsf{c}_{\mathsf{ahu}}(n_u^{i-1}) \mid n_u^{i-1} \in \mathrm{chi}(n_v^i)\}\!\}) \\ &= f(\mu(v), \{\!\{\mathsf{c}_{\mathsf{ahu}}(n_u^{i-1}) \mid u \in N(v)\}\!\}). \end{aligned} \tag{3}$$

By implementing the function f using a suitable neural architecture and replacing its codomain with embeddings in $\mathbb{R}^d$, we readily obtain a GNN based on our canonization approach. The key difference to standard GNNs is that the first component of the pair in Eq. (3) is the initial vertex feature instead of the embedding from the previous iteration. Utilizing the technique proposed by Xu et al. [36] and replacing the first addend in Eq. (1) with the initial embedding, we formulate the *unfolding tree canonization GNN*

$$x_i(v) = \mathrm{MLP}_i\left((1+\epsilon_i)\cdot x_0(v) + \sum_{u \in N(v)} x_{i-1}(u)\right). \tag{4}$$

It is established that MPNNs cannot distinguish two vertices with the same WL color or unfolding tree. Given that the function $\mathsf{c}_{\mathsf{ahu}}(n_v^i)$ uniquely represents the unfolding tree for an injective function f, realizable by Eq. (4) [36], we infer the following proposition.

Proposition 1. *Unfolding tree canonization GNNs, as defined in Eq. (4), are as expressive as GIN, as defined in Eq. (1).*

Despite the equivalence in expressivity, the canonization-based approach avoids redundancy since $x_{i-1}(v)$ represents the entire unfolding tree rooted at v of height $i-1$, while using the initial vertex features $x_0(v)$ is sufficient. We proceed by investigating redundancy within unfolding trees themselves.

GNNs via Neighborhood Tree Canonization. We leverage the concept of neighborhood trees to manage redundancy in unfolding trees.[2] A k-redundant neighborhood tree (k-NT) $T_{i,k}^v$ is derived from the unfolding tree F_i^v by removing all subtrees with roots that occurred more than k levels before (seen from root to leaves). Here, $\mathrm{depth}(v)$ denotes the length of the path from v to the root, and $\phi(v)$ denotes the original vertex in $V(G)$ represented by v in the tree.

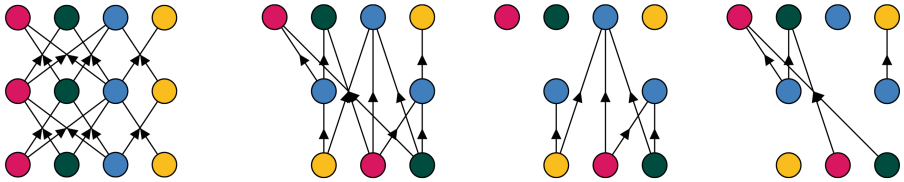
Definition 1 (k-redundant Neighborhood Tree). *For $k \geq 0$, the k-*redundant neighborhood tree *(k-NT) of a vertex $v \in V(G)$ with height i, denoted by $T_{i,k}^v$, is defined as the subtree of the unfolding tree F_i^v induced by the nodes $u \in V(F_i^v)$ satisfying*

$$\forall w \in V(F_i^v)\colon \phi(u) = \phi(w) \Rightarrow \mathrm{depth}(u) \leq \mathrm{depth}(w) + k.$$

Figures 2 and 3 provide examples of unfolding and neighborhood trees. It is worth noting that for $k \geq i$ the k-NT is equivalent to the WL unfolding tree.

[2] In a parallel work, neighborhood trees were investigated for approximating the graph edit distance [7].

Fig. 3. Graph G and its 0-NTs $T^v_{2,0}$ for all $v \in V(G)$.



(a) Merged unf. trees (b) Merged 0-NTs (c) $\mathcal{E}_1$ (d) $\mathcal{E}_2$

Fig. 4. Computation DAGs for unfolding (a) and 0-NTs (b) of height 2 of graph G. And edges in the different layers of the merge DAG of 0-NTs (c), (d).

We can directly apply the neural tree canonization technique to neighborhood trees. However, a simplifying expression based on the neighbors in the input graph, as given by Eq. (3) for unfolding trees, is not possible for neighborhood trees. Therefore, we explore techniques to systematically exploit computational redundancy.

4.2 Removing Computational Redundancy

The computation DAG of an MPNN involves the embedding of a set of trees representing the vertex neighborhoods of a single or multiple graphs. Results computed for one tree can be reused for others by identifying isomorphic substructures, thereby minimizing computational redundancy. We first describe how to merge trees in a general context and then discuss its application to unfolding and neighborhood trees.

Merging Trees Into a DAG. The neural tree canonization approach developed in the last section can be directly applied to DAGs. Given a DAG D, it computes an embedding for each node n in D that represents the tree F_n obtained by recursively following its children, similar as in unfolding trees, cf. Sect. 3. Since D is acyclic, the height of F_n is bounded. A detailed description of a neural architecture is postponed to Sect. 4.3.

Given a set of trees $\mathcal{T} = \{T_1, \ldots, T_n\}$, a *merge DAG* of $\mathcal{T}$ is a pair (D, ξ), where D is a DAG, $\xi\colon \{1, \ldots n\} \to V(D)$ is a mapping, and for all $i \in \{1, \ldots, n\}$ we have $T_i \simeq F_{\xi(i)}$. The definition guarantees that the neural tree canonization approach applied to the merge DAG produces the same result for the nodes in the DAG as for the nodes in the original trees. A trivial merge DAG is the disjoint union of the trees with $\xi(i) = r(T_i)$. However, depending on the structure of the given trees, we can identify the subtrees they have in common and represent them only once, such that two nodes of different trees share the same child, resulting in a DAG instead of a forest.

We propose an algorithm that builds a merge DAG by successively adding trees to an initially empty DAG, creating new nodes only when necessary. Our approach maintains a canonical labeling for each node of the DAG and computes a canonical labeling for each node of the tree to be added using the AHU algorithm. Then, the tree is processed starting at the root. If the canonical labeling of the root is present in the DAG, then the algorithm terminates. Otherwise, the subtrees rooted at its children are inserted into the DAG by recursive calls. Finally, the root is created and connected to the representatives of its children in the DAG. We introduce a node labeling $L\colon V_T \to \mathcal{O}$ used for tree canonization, where $V_T = \bigcup_{i=1}^{n} V(T_i)$ and $\mathcal{O}$ an arbitrary set of labels, refining the original node attributes, i.e., $L(u) = L(v) \Rightarrow \mu(u) = \mu(v)$ for all u, v in V_T. When $\mathcal{O}$ consists of integers from the range 1 to $|V_T|$, the algorithm runs in $O(|V_T|)$ time. When two siblings that are the roots of isomorphic subtrees are merged, this leads to parallel edges in the DAG. Parallel edges can be avoided by using a labeling satisfying $L(u) = L(v) \Rightarrow \mu(u) = \mu(v) \wedge p(u) \neq p(v)$ for all u, v in V_T.

Unfolding trees and k-NTs can grow exponentially in size with increasing height. However, this is not case for merge DAGs obtained by the algorithm described above, as we will show below. Moreover, we can directly generate DAGs of size $O(m \cdot (k+1))$ representing individual k-NTs with unbounded height in a graph with m edges (see Bause et al. [7] for details on generating compact DAGs).

Merging Unfolding Trees. Merging the unfolding trees of a graph with the labeling $L = \phi$ leads to the computation DAG of GNNs. Figure 4a shows the computation DAG for the graph from Fig. 1. The roots in this DAG correspond to the representation of the vertices after aggregating information from the lower layers. Each vertex occurs once at every layer of the DAG, and the links between any two consecutive layers are given by the adjacency matrix of the original graph. While this allows computation based on the adjacency matrix widely-used for MPNNs, it involves the encoding of redundant information. Our method has the potential to compress the computation DAG further by using the less restrictive labeling $L = \mu$, leading to a DAG where at layer i all vertices u, v with $c^{(i)}_{\mathsf{wl}}(u) = c^{(i)}_{\mathsf{wl}}(v)$ are represented by the same node. This compression appears particularly promising for graphs with symmetries.

Merging Neighborhood Trees. When merging k-redundant neighborhood trees in the same way using the labeling $L = \mu$ (or $L = \phi$ to avoid parallel edges), it results in a computation DAG with a more irregular structure, as illustrated in Fig. 4b. Firstly, there might be multiple nodes at the same level representing the same original vertex. Secondly, the adjacency matrix of the original graph cannot be used to propagate the information. A straightforward upper bound on the size of the merge DAG for a graph with n nodes and m edges is $O(nmk + nm)$.

We apply the neural tree canonization approach to the merge DAG in a bottom-up fashion, starting from the leaves and progressing to the roots. Each

edge is used exactly once in this computation. Let $D = (\mathcal{V}, \mathcal{E})$ be a merge DAG. The nodes can be partitioned based on their height, leading to $\mathcal{L}_i = \{v \in \mathcal{V} \mid \mathrm{hgt}(v) = i\}$. This induces an edge partition $\mathcal{E}_i = \{uv \in \mathcal{E} \mid v \in \mathcal{L}_i\}$, where all edges with the same end node v are in the same layer, and all incoming edges of children of v belong to a previous layer. Note that, since $\mathcal{L}_0$ contains all leaves of the DAG, there is no $\mathcal{E}_0$. Figures 4c and 4d depict the edge sets $\mathcal{E}_1$ and $\mathcal{E}_2$ for the example merge DAG illustrated in Fig. 4b.

4.3 Non-redundant Neural Architecture (DAG-MLP)

We propose a neural architecture to compute embeddings for nodes in a merge DAG, allowing to retrieve embeddings of the contained trees from their roots. The process involves a preprocessing step that transforms the node labels, using MLP_0 to map them to an embedding space of fixed dimensions. Subsequently, an MLP_i is used to process nodes at each layer $\mathcal{L}_i$.

$$\begin{aligned} \mu'(v) &= \mathrm{MLP}_0\left(\mu(v)\right), && \forall v \in \mathcal{V} \\ x(v) &= \mu'(v), && \forall v \in \mathcal{L}_0 \\ x(v) &= \mathrm{MLP}_i\left((1+\epsilon_i)\cdot\mu'(v) + \sum_{\forall u\colon uv \in \mathcal{E}_i} x(u)\right), && \forall v \in \mathcal{L}_i, i \in \{1,\ldots,n\} \end{aligned}$$

The DAG-MLP can be computed through iterated matrix-vector multiplication analogous to standard GNNs. Let $\mathbf{L}_i$ be a square matrix with ones on the diagonal at position j if $v_j \in \mathcal{L}_i$, and zeros elsewhere. Let $\mathbf{E}_i$ represent the adjacency matrix of $(\mathcal{V}, \mathcal{E}_i)$, and let $\mathbf{F}$ denote the node features of $\mathcal{V}$, corresponding to the initial node labels. The transformed features $\mathbf{F}'$ are obtained through MLP_0, and $\mathbf{X}^{[i]}$ represents the updated embeddings at layer i of the DAG.

$$\begin{aligned} \mathbf{F}' &= \mathrm{MLP}_0\left(\mathbf{F}\right), \quad \mathbf{X}^{[0]} = \mathbf{L}_0\mathbf{F}', \\ \mathbf{X}^{[i]} &= \mathrm{MLP}_i\left((1+\epsilon_i)\cdot\mathbf{L}_i\mathbf{F}' + \mathbf{E}_i\mathbf{X}^{[i-1]}\right) + \mathbf{X}^{[i-1]} \end{aligned}$$

In the above equation, MLP_i is applied to the rows associated with nodes in $\mathcal{L}_i$. The embeddings $\mathbf{X}^{[i]}$ are initialized to zero for inner nodes and computed level-wise. To preserve embeddings from all previous layers, we add $\mathbf{X}^{[i-1]}$ during the computation of $\mathbf{X}^{[i]}$. Suppose the merge DAG (D, ξ) contains the trees $\{T_1, \ldots, T_n\}$. We obtain a node embedding $\mathbf{X}^{[n]}_{\xi(i)}$ for each tree T_i with $i \in \{1, \ldots, n\}$. This approach allows for obtaining the final embedding for a vertex by using a single tree (Fixed Single-Height) or combining trees of different heights, for example all NTs of size up to a certain maximum (Combine Heights).

Figure 5 shows the DAG-MLP architecture for graph-level tasks. Vertex features are transformed using MLP_0. Messages propagate through the DAG according to $\mathcal{E}_i$. After n layers, all node embeddings have been computed and stored in $\mathbf{X}^{[n]}$. Readouts extract embeddings from k-NTs of varying heights. Pooling operations are applied to each layer's output, and the pooled outputs are averaged and processed by a final MLP for prediction.

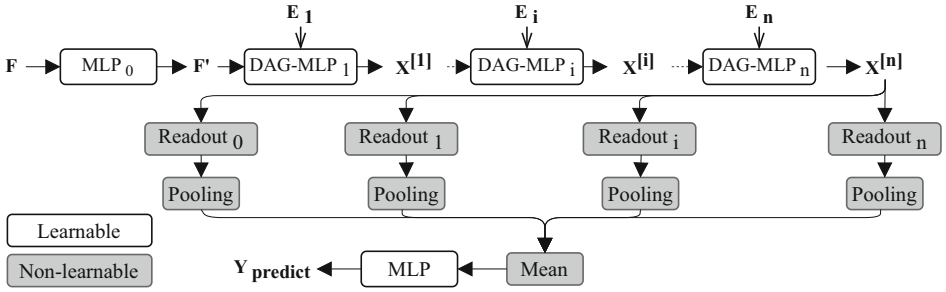


Fig. 5. DAG-MLP architecture for graph-level tasks with n layers. Each layer includes a DAG-MLP, readout, and pooling operations, processing the input $\mathbf{F}$.

4.4 Expressivity of k-NTs

We investigate how expressive k-NTs are compared to unfolding trees. While it is evident that k-NTs are a node invariant, providing the same result for nodes that can be mapped to each other by an isomophism or automorphism, they might also produce the same results for nodes that cannot. This means that, similar to unfolding trees, they are not a complete node invariant.

We show that there are vertices that 1-WL cannot distinguish, but k-NTs can, and vice versa, proving that both methods are incomparable regarding their expressivity on node level. We further conjecture that, while k-NTs cannot distinguish certain vertices that 1-WL can, they can still distinguish the graphs containing such vertices, making k-NTs more expressive on the graph level.

Theorem 1. *The expressivity of k-NT and unfolding trees is incomparable, i.e.,*

$$1.\ \exists u, v\colon F^u_\infty = F^v_\infty \wedge T^u_{\infty,k} \neq T^v_{\infty,k}$$
$$2.\ \exists u, v\colon F^u_\infty \neq F^v_\infty \wedge T^u_{\infty,k} = T^v_{\infty,k}$$

Proof. We prove the statement by giving concrete examples. Figure 6 gives an example for 1.: the Weisfeiler-Leman algorithm is unable to distinguish the two graphs, indicating identical unfolding trees of the vertices. However, for any k the k-NT of the vertices will differ for $h \geq k + 2$. Figure 7 gives an example for 2. (for 1-NTs): the unfolding trees of the two marked vertices differ, while the 1-NTs are identical.

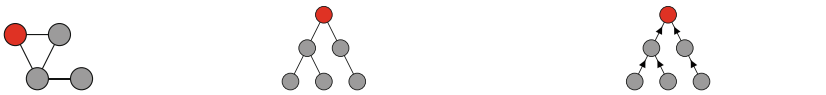
We have shown that on node level, the expressivity of k-NTs and unfolding trees is incomparable. However, the examples, where k-NTs fail to distinguish nodes that 1-WL can, can actually be distinguished on the graph level. This arises from the fact, that the graphs have a different number of vertices and the k-NTs of the other nodes differ.

4.5 Theoretical Analysis and Comparison to Related Work

We provide a detailed analysis of the computational complexity and expressivity of our approach compared to related work and formally prove, that our method can mitigate oversquashing better than comparable approaches.

(a) Hexagon (b) Two triangles (c) Unfolding trees (d) 1-NTs

Fig. 6. Two graphs (a), (b) that cannot be distinguished by unfolding trees, but by k-NTs. Figure (c) shows the unfolding tree F_3, which is the same for all vertices of both graphs, while (d) shows the 1-NTs of the vertices in the hexagon (left) and the triangle (right).



(a) G_1 (b) G_2 (c) 1-NTs of red vertices

Fig. 7. Two graphs (a), (b) in which the red vertices can be distinguished by unfolding trees, but not by k-NTs. The 1-NTs of the red vertices (c) are the same. However, G_1 and G_2 can be distinguished by their multisets of 1-NTs. (Color figure online)

Computational Complexity and Expressivity. The DAG representing the k-NT of a single vertex has a size in $O(mk+m)$, where m is the number of edges in the graph. The lexicographic encoding and merging of k-NTs to generate the DAG can be done in time linear in its size. A trivial upper bound on the size of the merge DAG of a graph with n nodes and m edges is $O(nmk+nm)$. Overall, this means that preprocessing can be done in $O(nmk)$ time, where k can be considered constant. For 0- and 1-NTs, we obtain time $O(nm)$.

Table 1 compares the complexity and expressivity of our method to related work. Understanding and relating the expressivity of different approaches is non-trivial. In the case of PathNet, its expressivity concerning the WL-hierarchy remains unexplored. PathNN-SP+ has been shown to be more expressive than 1-WL. While it is claimed that RFGNNs are maximally expressive, the proof claiming higher expressivity on node level as presented in Chen et al. [9, Lemma 7] is not correct (rather it is incomparable on node level), which was shown in Bause et al. [6]. Consequently, it remains uncertain, whether RFGNN is strictly more expressive on graph level. PathNN-SP [27] states that it can only disambiguate graphs at least as well as 1-WL and is not strictly more powerful. Due to sampling, isomorphic graphs could be mapped to different representations, indicating that it is not a graph invariant.

RFGNN and PathNN-SP+ involve enumerating all possible paths and all shortest paths, respectively. Our approach avoids this redundancy by not explicitly building NTs, but instead generate DAGs, resulting in a much more compact representation (refer to Bause et al. [7] for an algorithm on how to generate compact DAGs). We investigate the theoretical connection of our method and RFGNN in terms of relative influence, and show that 0- and 1-NTs can address the oversquashing problem more effectively.

Oversquashing. Several authors [8,9,14,34,37] developed and refined techniques to measure the influence of a vertex v with initial vertex feature $\mathbf{x}_v$ on the output $\mathbf{h}_u^{(l)}$ of a vertex u after layer l by the Jacobian $\partial \mathbf{h}_u^{(l)} / \partial \mathbf{x}_v$. Following Chen et al. [9, Lemma 3], the relative influence of a vertex v on a vertex u in an MPNN is

$$I(v,u) = \mathbb{E}\left(\frac{\partial \mathbf{h}_u^{(l)} / \partial \mathbf{x}_v}{\sum_{w \in V} \partial \mathbf{h}_u^{(l)} / \partial \mathbf{x}_w} \right) = \frac{[\hat{A}^l]_{u,v}}{\sum_{w \in V} [\hat{A}^l]_{u,w}},$$

where $\hat{A} = A + I$ is the adjacency matrix of G with added self-loops, and $[\hat{A}^l]_{u,v}$ is the number of walks of length l from u to v (and vice versa) in G with added self-loops. Oversquashing occurs when $I(v,u)$ becomes small, indicating that only a small fraction of walks of length up to l ending at u start at v.

This idea can easily be linked to the trees underlying our work. Consider the unfolding tree F_l^u of u with height l. It follows from its construction that there is a bijection between walks of length at most l ending at u in G and paths in F_l^u from some node to the root (see [18] for details on unfolding trees and walks). Therefore, pruning the unfolding tree has an effect on walk counts and, thus, on the relative influence. Consider the example in Fig. 1 and let u be the red vertex (upper left) and v the yellow vertex (lower right). We obtain a relative influence of $I_{\mathrm{MPNN}}(v,u) = \frac{1}{8}$ for unfolding trees, and $I_{0\mathrm{NT}}(v,u) = \frac{1}{4}$ for 0-NTs, showing that NTs have the potential to reduce oversquashing.

We formally show that our method is less susceptible to oversquashing than MPNNs and RFGNN [9]. Consider a vertex v and a vertex u with shortest-path distance of l. To pass information from v to u, at least l layers are required. Comparing the unfolding tree underlying MPNNs, the 0- and 1-NT, and the TPT used in RFGNN, all of height l, reveals that the vertex v occurs in the last level only, i.e., as a leaf of the tree, and the number of occurrences of v is equal in all trees, since all walks and simple paths of length l reaching v are shortest paths. Hence, the numerator of the relative influence is equal for all methods. However, since 0- and 1-NTs are subtrees of unfolding trees, and 0-NTs/1-NTs are subtrees of TPTs (they contain only shortest paths/some simple paths, instead of all simple paths), the total number of nodes in the trees, i.e., walks contributing to the denominator of the relevant information can be compared, obtaining

$$I_{\mathrm{MPNN}}(v,u) \leq I_{\mathrm{TPT}}(v,u) \leq I_{1\mathrm{NT}}(v,u) \leq I_{0\mathrm{NT}}(v,u).$$

This theoretical analysis shows that our proposed method offers advantages in mitigating oversquashing, leveraging the formalization developed in recent papers. Additionally, it establishes a theoretical connection between the proposed approach and RFGNN [9], highlighting that our method more effectively addresses the oversquashing problem.

To summarize, our method is the first to address both types of redundancy in GNNs, informational and computational redundancy, with polynomial running time, and can mitigate oversquashing better than comparable approaches.

5 Experimental Evaluation

We assess the performance of DAG-MLP with k-NTs on a range of synthetic [1,29] and real-world datasets [10,12,25,28,32].[3]

Experimental Setup. For synthetic datasets, we determine the number of layers in DAG-MLP based on the average graph diameter, ensuring effective aggregation during message propagation. The embeddings at each layer are obtained using readouts, concatenated, and then fed through two learnable linear layers for prediction. For TUDataset, we use the 10-fold splits proposed by Errica et al. [11], allowing a grid search for optimal hyper-parameters. The architecture for combined heights involves using each "readout$_i$" to extract the embeddings for each layer, with mean of the average-pooled embeddings being passed to a final MLP layer responsible for prediction. For the fixed single-height architecture, only the last readout is used, pooled, and passed to the final MLP layer.

Graph Classification. Table 2 presents the results on synthetic expressivity datasets. Our hypothesis that NTs are more expressive than GIN on graph level is experimentally validated, but a theoretical proof remains future work.

In Table 3, we investigate the impact of the parameter k and the number of layers l on the accuracy for the EXP-Class dataset. Cases where $k > l$ can be disregarded, as the computation for NTs remains the same as when $k = l$. Empirically, 0- and 1-NTs yield the highest accuracy. This observation aligns with our discussions on expressivity in Sect. 4.4. The decrease in accuracy with increasing k indicates that information redundancy leads to oversquashing. We investigate this theoretically in Sect. 4.5.

For TUDataset, we report the accuracy compared to related work in Table 4. We report only the best results for the different parameter combinations reported in Michel et al. [27], and the best result for our different combine methods. We group the methods by their time complexity. Note that, while PathNN performs well on ENZYMES and PROTEINS, the time complexity of this method is exponential. Therefore, we also highlight the best method with polynomial time complexity. For IMDB-B and IMDB-M, which have small diameters, k-NTs outperform all other methods. For ENZYMES a variant of our approach achieves the best result among the approaches with non-exponential time complexity, and k-NTs lead to a significant improvement over GIN.

Node Classification. We investigate the performance of our approach on node classification datasets. These datasets differ regarding their homophily ratio, i.e., the fraction of edges in a graph that connect vertices with the same class label [40]. Heterophily tasks are particularly challenging for standard GNNs [40] as they require capturing the structure of neighborhoods instead of "averaging" over the neighboring features. In Table 5 we present results from Giraldo et

[3] The implementation is available at github.com/samirmoustafa/k-redundancyGNNs.

Table 2. Average classification accuracy for EXP-Class and CSL across k-folds (4-folds and 5-folds), and the number of indistinguishable pairs of graphs in EXP-Iso. Best results are highlighted in gray, best results from methods with polynomial time complexity are highlighted in **bold**.

Model	EXP-Class ↑	EXP-Iso ↓	CSL ↑
GIN [36]	50.0 ± 0.0	600	10.0 ± 0.0
3WLGNN [24]	**100.0 ± 0.0**	**0**	97.8 ± 10.9
PathNN-$\mathcal{SP}^+$ [27]	100.0 ± 0.0	0	100.0 ± 0.0
PathNN-$\mathcal{AP}$ [27]	100.0 ± 0.0	0	100.0 ± 0.0
DAG-MLP (0-NTs)	**100.0 ± 0.0**	**0**	**100.0 ± 0.0**
DAG-MLP (1-NTs)	**100.0 ± 0.0**	**0**	**100.0 ± 0.0**

Table 3. Average accuracy for DAG-MLP using 4-fold cross-validation on EXP-Class [1], evaluated with varying number of layers.

k-NTs	1 layer	2 layers	3 layers	4 layers	5 layers	6 layers
0-NTs	51.1 ± 1.6	57.5 ± 6.6	91.7 ± 11.6	99.7 ± 0.3	**100.0 ± 0.0**	**100.0 ± 0.0**
1-NTs	50.1 ± 0.2	58.9 ± 4.6	59.4 ± 5.7	99.6 ± 0.5	99.9 ± 0.2	**100.0 ± 0.0**
2-NTs	–	52.6 ± 3.4	54.9 ± 5.3	52.4 ± 3.8	97.6 ± 1.9	**100.0 ± 0.0**
3-NTs	–	–	56.2 ± 5.7	51.1 ± 1.9	52.4 ± 4.1	87.1 ± 21.4
4-NTs	–	–	–	50.1 ± 0.2	50.6 ± 1.0	50.4 ± 0.7
5-NTs	–	–	–	–	50.4 ± 0.7	50.0 ± 0.0
6-NTs	–	–	–	–	–	53.2 ± 5.2

al. [15] including the state-of-the-art graph rewiring technique SJLR combined with SGC and GCN, which performs best in the evaluation. We also performed experiments with GIN and DAG-MLP, using the same data splits as Giraldo et al. [15] to ensure a fair comparison. We report the best results for l layers with $l \in \{2, 3, 4\}$ and four different combine methods for GIN and DAG-MLP.

DAG-MLP outperforms GIN on heterophily datasets (those with low homophily ratio), while GIN performs better on homophily ones, indicating that neighborhood trees can capture the relevant neighborhood structure more accurately than unfolding trees used by GIN. Additionally, our method outperforms SJLR on heterophily datasets Texas and Wisconsin by a large margin.

6 Conclusion

We introduce a neural tree canonization technique and combine it with neighborhood trees, which are pruned versions of unfolding trees used by standard MPNNs. By merging trees in a DAG, we create compact representations that

Table 4. Classification accuracy (± standard deviation) over 10-fold cross-validation on the datasets from TUDataset, taken from Michel et al. [27]. Best results highlighted in gray, best results from methods with polynomial time complexity highlighted in **bold**. "-": not applicable, "NA": not available.

	Model	IMDB-B	IMDB-M	ENZYMES	PROTEINS
Linear	GIN [36]	71.2 ± 3.9	48.5 ± 3.3	59.6 ± 4.5	73.3 ± 4.0
	GAT [35]	69.2 ± 4.8	48.2 ± 4.9	49.5 ± 8.9	70.9 ± 2.7
	SPN ($l = 1$) [2]	NA	NA	67.5 ± 5.5	71.0 ± 3.7
	SPN ($l = 5$) [2]	NA	NA	69.4 ± 6.2	**74.2 ± 2.7**
Exp	PathNet [33]	70.4 ± 3.8	49.1 ± 3.6	69.3 ± 5.4	70.5 ± 3.9
	PathNN-$\mathcal{P}$ [27]	72.6 ± 3.3	50.8 ± 4.5	73.0 ± 5.2	75.2 ± 3.9
	PathNN-$\mathcal{SP}^+$ [27]	-	-	70.4 ± 3.1	73.2 ± 3.3
Ours	DAG-MLP (0-NTs)	**72.9 ± 5.0**	50.2 ± 3.2	67.9 ± 5.3	70.1 ± 1.7
	DAG-MLP (1-NTs)	72.4 ± 3.8	**51.3 ± 4.4**	**70.6 ± 5.5**	70.2 ± 3.4

Table 5. Classification accuracy (± standard deviation) on node classification tasks (GCN, SJLR-GCN, SGC and SJLR-GCN taken from Giraldo et al. [15]).

Model	Texas	Wisconsin	Cornell	Cora	CiteSeer	PubMed
Homophily ratio	0.11	0.21	0.3	0.8	0.74	0.8
GCN	58.05 ± 0.9	52.10 ± 0.9	67.34 ± 1.5	81.81 ± 0.2	68.35 ± 0.3	78.25 ± 0.3
SJLR-GCN	60.13 ± 0.8	55.16 ± 0.9	71.75 ± 1.5	**81.95 ± 0.2**	**69.50 ± 0.3**	**78.60 ± 0.3**
SGC	56.69 ± 1.7	47.90 ± 1.7	53.40 ± 2.1	76.90 ± 1.3	67.45 ± 0.8	71.79 ± 2.1
SJLR-SGC	58.40 ± 1.4	55.42 ± 0.9	67.37 ± 1.6	81.24 ± 0.7	68.39 ± 0.6	76.28 ± 0.9
GIN	73.78 ± 6.0	71.76 ± 5.1	60.81 ± 8.5	76.76 ± 1.4	64.49 ± 1.5	76.46 ± 1.1
DAG-MLP (0-NTs)	**85.68 ± 4.8**	81.35 ± 4.1	79.02 ± 6.8	74.01 ± 2.0	60.55 ± 3.6	75.33 ± 1.1
DAG-MLP (1-NTs)	80.54 ± 6.0	**81.62 ± 3.4**	**79.41 ± 4.6**	74.54 ± 1.4	61.09 ± 1.5	75.53 ± 1.1

serve as the foundation for our neural architecture termed DAG-MLP. It inherits the advantageous properties of the GIN architecture, while being more expressive than 1-WL on many graphs. Notably, our method is only less expressive on node level for specific examples. Our work contributes general techniques for constructing compact computation DAGs for tree structures that encode node neighborhoods. This exploration reveals a complex interplay between information redundancy, computational redundancy, and expressivity. The delicate balance of these factors is an avenue for future work.

Acknowledgments. We would like to thank Christian Permann for his contribution to the conception of k-redundant neighborhood trees and their efficient generation. This work was supported by the Vienna Science and Technology Fund (WWTF) [10.47379/VRG19009]. The computational results presented have been achieved in part using the Vienna Scientific Cluster (VSC).

References

1. Abboud, R., Ceylan, I.I., Grohe, M., Lukasiewicz, T.: The surprising power of graph neural networks with random node initialization. In: IJCAI (2021)
2. Abboud, R., Dimitrov, R., Ceylan, İ.İ.: Shortest path networks for graph property prediction. LoG (2022)
3. Abu-El-Haija, S., Perozzi, B., et al.: MixHop: higher-order graph convolutional architectures via sparsified neighborhood mixing. In: ICML (2019)
4. Aho, A.V., Hopcroft, J.E., Ullman, J.D.: The Design and Analysis of Computer Algorithms. Addison-Wesley (1974)
5. Alon, U., Yahav, E.: On the bottleneck of graph neural networks and its practical implications. In: ICLR (2021)
6. Bause, F., Moustafa, S., Langguth, J., Gansterer, W.N., Kriege, N.M.: On the two sides of redundancy in graph neural networks. CoRR abs/2310.04190 (2023)
7. Bause, F., Permann, C., Kriege, N.: Approximating the graph edit distance with compact neighborhood representations. In: ECML/PKDD (2024)
8. Black, M., Wan, Z., Nayyeri, A., Wang, Y.: Understanding oversquashing in GNNs through the lens of effective resistance. In: ICML (2023)
9. Chen, R., Zhang, S., U, L.H., Li, Y.: Redundancy-free message passing for graph neural networks. In: NeurIPS (2022)
10. Craven, M.W., et al.: Learning to extract symbolic knowledge from the world wide web. In: AAAI/IAAI (1998)
11. Errica, F., Podda, M., Bacciu, D., Micheli, A.: A fair comparison of graph neural networks for graph classification. In: ICLR (2020)
12. Giles, C.L., Bollacker, K.D., Lawrence, S.: Citeseer: an automatic citation indexing system. Digital library (1998)
13. Gilmer, J., Schoenholz, S.S., Riley, P.F., Vinyals, O., Dahl, G.E.: Neural message passing for quantum chemistry. In: ICML (2017)
14. Giovanni, F.D., Giusti, L., Barbero, F., Luise, G., Lio, P., Bronstein, M.M.: On over-squashing in message passing neural networks: The impact of width, depth, and topology. In: ICML (2023)
15. Giraldo, J., Skianis, K., Bouwmans, T., Malliaros, F.: On the trade-off between over-smoothing and over-squashing in deep graph neural networks. In: CIKM (2023)
16. Jegelka, S.: Theory of graph neural networks: Representation and learning. CoRR abs/2204.07697 (2022)
17. Jia, Z., Lin, S., Ying, R., You, J., Leskovec, J., Aiken, A.: Redundancy-free computation for graph neural networks. In: KDD (2020)
18. Kriege, N.M.: Weisfeiler and Leman go walking: random walk kernels revisited. In: NeurIPS (2022)
19. Kriege, N.M., Giscard, P.L., Wilson, R.C.: On valid optimal assignment kernels and applications to graph classification. In: NIPS (2016)
20. Li, P., Wang, Y., Wang, H., Leskovec, J.: Distance encoding: Design provably more powerful neural networks for graph representation learning. In: NeurIPS (2020)
21. Li, Q., Han, Z., Wu, X.: Deeper insights into graph convolutional networks for semi-supervised learning. In: AAAI (2018)
22. Liu, M., Gao, H., Ji, S.: Towards deeper graph neural networks. In: KDD (2020)
23. Mahé, P., Ueda, N., Akutsu, T., Perret, J.L., Vert, J.P.: Extensions of marginalized graph kernels. In: ICML (2004)

24. Maron, H., Ben-Hamu, H., Serviansky, H., Lipman, Y.: Provably powerful graph networks. In: NeurIPS (2019)
25. McCallum, A., Nigam, K., Rennie, J.D.M., Seymore, K.: Automating the construction of internet portals with machine learning. Information Retrieval (2000)
26. Mehlhorn, K., Sanders, P.: Algorithms and Data Structures: The Basic Toolbox. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-77978-0
27. Michel, G., Nikolentzos, G., Lutzeyer, J., Vazirgiannis, M.: Path neural networks: expressive and accurate graph neural networks. In: ICML (2023)
28. Morris, C., Kriege, N.M., Bause, F., Kersting, K., Mutzel, P., Neumann, M.: TUDataset: a collection of benchmark datasets for learning with graphs. In: ICML GRL+ Workshop (2020)
29. Murphy, R., Srinivasan, B., Rao, V., Ribeiro, B.: Relational pooling for graph representations. In: ICML (2019)
30. Nikolentzos, G., Chatzianastasis, M., Vazirgiannis, M.: Weisfeiler and Leman go hyperbolic: Learning distance preserving node representations. In: AISTATS (2023)
31. Scarselli, F., Gori, M., Tsoi, A.C., Hagenbuchner, M., Monfardini, G.: Computational capabilities of graph neural networks. IEEE Trans. Neural Networks (2009)
32. Sen, P., Namata, G., Bilgic, M., Getoor, L., Gallagher, B., Eliassi-Rad, T.: Collective classification in network data. AI Mag. (2008)
33. Sun, Y., et al.: Beyond homophily: structure-aware path aggregation graph neural network. In: IJCAI (2022)
34. Topping, J., Giovanni, F.D., Chamberlain, B.P., Dong, X., Bronstein, M.M.: Understanding over-squashing and bottlenecks on graphs via curvature. In: ICLR (2022)
35. Veličković, P., Cucurull, G., Casanova, A., Romero, A., Lio, P., Bengio, Y.: Graph attention networks. In: ICLR (2018)
36. Xu, K., Hu, W., Leskovec, J., Jegelka, S.: How powerful are graph neural networks? In: ICLR (2019)
37. Xu, K., Li, C., Tian, Y., Sonobe, T., Kawarabayashi, K., Jegelka, S.: Representation learning on graphs with jumping knowledge networks. In: ICML (2018)
38. Yang, Y., Wang, X., Song, M., Yuan, J., Tao, D.: SPAGAN: shortest path graph attention network. In: IJCAI (2019)
39. You, J., Selman, J.M.G., Ying, R., Leskovec, J.: Identity-aware graph neural networks. In: AAAI (2021)
40. Zhu, J., Yan, Y., Zhao, L., Heimann, M., Akoglu, L., Koutra, D.: Beyond homophily in graph neural networks: current limitations and effective designs. In: NeurIPS (2020)

A.6 Maximally Expressive GNNs for Outerplanar Graphs

Authors Franka Bause*, Fabian Jogl*, Patrick Indri, Tamara Drucks, David Penz, Nils M. Kriege, Thomas Gärtner, Pascal Welke, Maximilian Thiessen.

Publication Venue Published in *Transactions on Machine Learning Research* (2025).

DOI/URL https://openreview.net/pdf?id=XxbQAsxrRC

Authors Contributions

Franka Bause* found the existing theoretical basis for the paper in the literature search, developed the graph transformations together with FJ, developed and proved Theorems 1 and 2, and drafted, wrote, and revised the main parts of the manuscript.

Fabian Jogl* developed the graph transformations together with FB, implemented the graph transformations, conducted the experimental evaluation with the exception of the experiments on graph connectivity, and helped to draft and revise the manuscript.

Patrick Indri investigated the graph connectivity, wrote the corresponding section together with TD, and conducted the experiments on graph connectivity.

Tamara Drucks investigated the graph connectivity and wrote the corresponding section together with PI.

David Penz gave valuable suggestions and feedback throughout the project.

Nils M. Kriege gave valuable suggestions and feedback throughout the project, and helped to revise the manuscript.

Thomas Gärtner gave valuable suggestions and feedback throughout the project.

Pascal Welke gave valuable suggestions and feedback throughout the project, wrote the preprocessing code for the graph transformation, created the figures in the main paper, and helped to formalize the definition of the graph transformation and to revise the manuscript.

Maximilian Thiessen initiated the project, drafted the related work section, gave valuable suggestions and feedback throughout the project, and helped to revise the manuscript.

Reference in Thesis [BJI+25]

*Equal contribution.

Abstract

We propose a linear time graph transformation that enables the Weisfeiler-Leman (WL) algorithm and message passing graph neural networks (MPNNs) to be maximally expressive on outerplanar graphs. Our approach is motivated by the fact that most pharmaceutical molecules correspond to outerplanar graphs. Existing research predominantly enhances the expressivity of graph neural networks without specific graph families in mind. This often leads to methods that are impractical due to their computational complexity. In contrast, the restriction to outerplanar graphs enables us to encode the Hamiltonian cycle of each biconnected component in linear time. As the main contribution of the paper we prove that our method achieves maximum expressivity on outerplanar graphs. Experiments confirm that our graph transformation improves the predictive performance of MPNNs on molecular benchmark datasets at negligible computational overhead.

Published in Transactions on Machine Learning Research (12/2024)

Maximally Expressive Graph Neural Networks for Outerplanar Graphs

Franka Bause[*1,2], **Fabian Jogl**[*3,4], **Patrick Indri**[3], **Tamara Drucks**[3], **David Penz**[3,5], **Nils M. Kriege**[1,6], **Thomas Gärtner**[3], **Pascal Welke**[3], **Maximilian Thiessen**[3]

[1] *Faculty of Computer Science, University of Vienna, Vienna, Austria*
[2] *UniVie Doctoral School Computer Science, University of Vienna, Vienna, Austria*
[3] *Machine Learning Research Unit, TU Wien, Vienna, Austria*
[4] *Center for Artificial Intelligence and Machine Learning, Vienna, Austria*
[5] *Multimedia Mining and Search, Johannes Kepler University Linz, Linz, Austria*
[6] *Research Network Data Science, University of Vienna, Vienna, Austria*

{firstname.lastname}@{univie, tuwien}.ac.at

Reviewed on OpenReview: *https://openreview.net/forum?id=XxbQAsxrRC*

Abstract

We propose a *linear time* graph transformation that enables the Weisfeiler-Leman (WL) algorithm and message passing graph neural networks (MPNNs) to be maximally expressive on *outerplanar* graphs. Our approach is motivated by the fact that most pharmaceutical molecules correspond to outerplanar graphs. Existing research predominantly enhances the expressivity of graph neural networks without specific graph families in mind. This often leads to methods that are impractical due to their computational complexity. In contrast, the restriction to outerplanar graphs enables us to encode the Hamiltonian cycle of each biconnected component in linear time. As the main contribution of the paper we prove that our method achieves maximum expressivity on outerplanar graphs. Experiments confirm that our graph transformation improves the predictive performance of MPNNs on molecular benchmark datasets at negligible computational overhead.

1 Introduction

We study graph neural networks (GNNs) for the class of outerplanar graphs and devise a linear time preprocessing step that message passing graph neural networks (MPNNs) to distinguish all non-isomorphic outerplanar graph. Morris et al. (2019) and Xu et al. (2019) showed that MPNNs have limited *expressivity*, i.e., there exist non-isomorphic graphs on which each MPNN produces the same embedding. Such graphs exist even within the restricted class of outerplanar graphs (see Figure 1). This led to the development of GNNs that are more expressive than MPNNs, often called *higher-order* GNNs. However, the increase in expressivity usually comes with a significant increase in computational complexity. For example, k-GNNs (Morris et al., 2019) have a time complexity of $\Omega(|V|^k)$. Other higher-order GNNs count pattern graphs such as cliques (Bodnar et al., 2021b), cycles (Bodnar et al., 2021a;b), and general subgraphs (Bouritsas et al., 2022), which can take time exponential in the pattern size. However, for certain domains of interest, the graph structure can be exploited to build efficient higher-order GNNs. In this work, we focus on the pharmaceutical domain and on graphs that represent molecules. Over 92% to 97% of the graphs in widely used benchmark datasets in this domain are *outerplanar* (see Table 1). The properties of outerplanar graphs have been exploited by algorithms for graph mining (Horváth et al., 2010) and molecular similarity computation (Schietgat et al., 2013; Droschinsky et al., 2017), but not yet for GNNs. We focus on this

[*] Equal contribution.

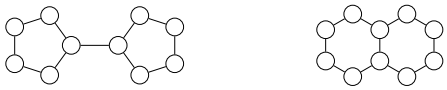


Figure 1: The molecules bicyclopentyl (left) and decalin (right) which correspond to outerplanar graphs that cannot be distinguished by MPNNs and 1-WL.

class of graphs and devise a linear time transformation that allows MPNNs to become maximally expressive on outerplanar graphs. This implies that, in principle, our architecture can solve any learning task on outerplanar graphs. While this does not mean that any given maximally expressive model will perform well in practice, our experiments show that our proposed transformation improves the predictive performance of several GNN architectures on multiple benchmark learning tasks.

We propose to decompose outerplanar graphs into biconnected outerplanar components and trees. Using the fact that each biconnected outerplanar component has a unique Hamiltonian cycle that can be computed in linear time, we split each component into two graphs corresponding to the directed Hamiltonian cycles, and prove that MPNNs are maximally expressive on biconnected outerplanar graphs transformed in this way. Taking advantage of the well-known fact that MPNNs are maximally expressive on labeled trees (Arvind et al., 2015; Kiefer, 2020), we extend our result to a linear time graph transformation called *Cyclic Adjacency Transform* (CAT) that works on all outerplanar graphs. We benchmark CAT with common MPNNs on a variety of molecular graph benchmarks and show that CAT consistently boosts the performance of MPNNs.

Main contributions. We propose CAT, a linear time graph transformation that renders MPNNs maximally expressive on outerplanar graphs. Experimentally, CAT consistently improves the performance of MPNNs with little increase in runtime.

2 Discussion and Related Work

Since the expressivity of MPNNs is bounded by the 1-WL algorithm (Morris et al., 2019; Xu et al., 2019), any pair of non-isomorphic graphs that cannot be distinguished by 1-WL will get mapped to the same embedding by any given MPNN. One such pair of graphs are the molecules decalin and bicyclopentyl (see Fig. 1). As these two graphs are outerplanar, it follows that MPNNs are not sufficiently expressive for outerplanar graphs. Furthermore, in the graph mining community it is well known that many pharmaceutical molecules are outerplanar (Horváth et al., 2006; Horváth & Ramon, 2010). Outerplanarity has also been discussed in the context of reconstruction with GNNs (Cotta et al., 2021). This motivates the need for GNNs that are highly expressive on outerplanar graphs. Outerplanar graphs have treewidth at most two (Bodlaender, 1998), and Kiefer (2020) showed that 3-WL is sufficiently expressive to distinguish all outerplanar graphs. Hence, any GNN that matches the expressivity of 3-WL, such as 3-IGN (Maron et al., 2019) or 3-GNN (Morris et al., 2019), is capable of solving our main goal of distinguishing all outerplanar graphs. However, a single iteration of a naive implementation of the 3-WL algorithm on a graph with n vertices is at least $\mathcal{O}(n^3)$ (Immerman & Lander, 1990; Kiefer, 2020), which can be infeasible even for medium-sized real-world graphs. Similarly, a single 3-GNN or 3-IGN layer runs in roughly $\mathcal{O}(n^3)$ time (Maron et al., 2019; Morris et al., 2019). Even when additionally restricting the graph class to *biconnected* outerplanar graphs, MPNNs are not sufficiently expressive. Furthermore, Zhang et al. (2023b) have shown that most GNNs cannot even detect simple properties associated with biconnectivity such as articulation vertices. They find that only their distance-based GNN and specific GNNs based on subgraphs (Bevilacqua et al., 2021; Frasca et al., 2022) are able to detect some of these properties. Again, these approaches have at least quadratic worst case runtime.

It is not straightforward to use outerplanarity to efficiently improve the expressivity of GNNs, as even finding a subgraph remains NP-hard for outerplanar graphs (Sysło, 1982). Thus, methods like the graph structural network (Bouritsas et al., 2022) that rely on counting subgraphs remain computationally expensive even on outerplanar graphs. Subgraph GNNs model graphs as a collection of subgraphs (Frasca et al., 2022), which usually requires a pre-processing with at least quadratic runtime, depending on the method used to extract

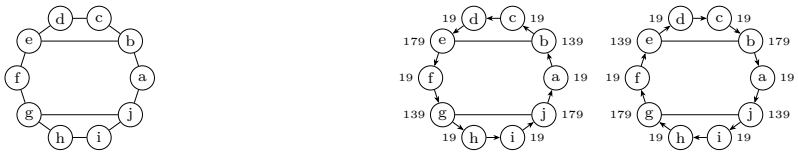


Figure 2: A graph and both directions of its directed Hamiltonian cycle. Nodes are annotated with their HALs, the distances on the Hamiltonian cycle to their neighbors (Colbourn & Booth, 1981).

subgraphs. For example, Node-delete (Bevilacqua et al., 2021) creates all subgraphs that are obtained by deleting a single node creating $\mathcal{O}(n^2)$ nodes in total. k-ego-net (Bevilacqua et al., 2021) extracts the k-hop neighborhood for each node which for $k \geq 2$ can create $\mathcal{O}(n^2)$ nodes in the worst case, for example for star graphs, which are also outerplanar.

Recently, Dimitrov et al. (2023) have proposed PlanE, a GNN architecture that can distinguish all planar graphs in $\mathcal{O}(n^2)$ time. As a consequence, PlanE is also able to distinguish all outerplanar graphs. However, this comes at the cost of (1) runtime, (2) flexibility, and—counter-intuitively—(3) expressivity. (1) PlanE requires a quadratic time pre-processing whereas our proposed CAT+MPNN runs in linear time. (2) PlanE is a GNN architecture while CAT is a graph transformation. This means, that CAT can be combined with any GNN while PlanE requires specialized GNN layers which are not easily combined with other architectures. (3) Without changes to PlanE's pre-processing, PlanE is incapable of operating on graphs with non-planar components whereas CAT still achieves at least 1-WL expressivity on such graphs. This leads to the counter-intuitive result that vanilla PlanE is incomparable in expressivity to CAT (see Prop. 2 and 3 in the Appendix).

Finally, while there exist many GNNs that are provably more expressive than WL, little is known about the precise class of graphs for which such GNNs are provably maximally expressive. Furthermore, proving an upper bound on the expressivity of an architecture is considered difficult and requires significant effort as demonstrated by Zhang et al. (2023a). In contrast, we identify outerplanar graphs as a large practical graph family that our proposed method CAT can distinguish.

3 Preliminaries

A *graph* $G = (V, E, \mu, \nu)$ consists of a set of nodes V, a set of edges $E \subseteq V \times V$ and attributes (also called features) for the nodes $\mu\colon V \to X$ and edges $\nu\colon E \to X$, respectively, where X is a set of arbitrary attributes. We refer to an edge from u to v by uv and in case of undirected graphs $uv = vu$. The *in-neighbors* of a node $u \in V$ are denoted by $N_{\text{in}}(u) = \{v \mid vu \in E\}$. The *out-neighbors* of a node $u \in V$ are denoted by $N_{\text{out}}(u) = \{v \mid uv \in E\}$ and in case of undirected graphs, $N_{\text{in}} = N_{\text{out}}$. In this paper, the input graphs are undirected and are transformed into directed ones. A graph $G' = (V', E', \mu', \nu')$ is a subgraph of a graph G, denoted by $G' \subseteq G$, iff $V' \subseteq V$, $E' \subseteq E$, $\forall v \in V' : \mu'(v) = \mu(v)$, and $\forall e \in E' : \nu'(e) = \nu(e)$. A (directed) cycle $(v_1, \ldots, v_k)$ is a sequence of $k \geq 3$ distinct nodes, with $\forall i \in \{1, \ldots, k-1\} : v_i v_{i+1} \in E$ and $v_k v_1 \in E$. A graph is *acyclic* if it does not contain a cycle. Given a graph G, we denote the shortest path distance between two nodes u and v by $d_G(u, v)$, or $d(u, v)$ if G is clear from the context. We denote the *diameter* of a graph G by $\Phi(G) = \max_{u,v \in V(G)} d(u, v)$.

A graph is *outerplanar* if it can be drawn in the plane without edge crossings and with all nodes belonging to the exterior face (see Appendix A for details). We call an undirected graph with at least three vertices *biconnected* if the removal of any single node does not disconnect the graph. A *biconnected component* is a maximal biconnected subgraph. We refer to the outerplanar biconnected components of a graph as *blocks*.

Two graphs G and H are isomorphic, if there exists a bijection $\psi\colon V(G) \to V(H)$, so that $\forall u, v \in V(G)\colon \mu(v) = \mu(\psi(v)) \wedge uv \in E(G) \Leftrightarrow \psi(u)\psi(v) \in E(H) \wedge \forall uv \in E(G)\colon \nu(uv) = \nu(\psi(u)\psi(v))$. We call ψ an isomorphism between G and H. An *in-tree* T is a directed, acyclic graph with a distinct *root*

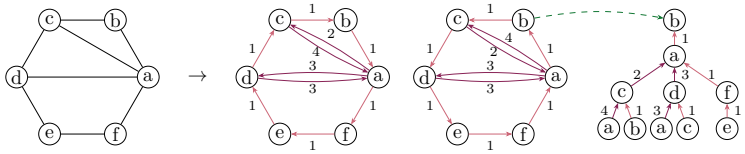


Figure 3: Biconnected outerplanar graph G, $\mathrm{CAT}^*(G)$, and unfolding tree of node b.

having no outgoing edges and all other nodes having one outgoing edge and for every node there is exactly one path from it to the root.

Weisfeiler-Leman. The 1-dimensional Weisfeiler-Leman algorithm (WL) assigns colors (usually represented by numbers) to nodes. The color of a node $v \in V(G)$ is updated iteratively according to $c_{i+1}(v) = h\left(c_i(v), \{\!\{(\nu(uv), c_i(u)) \mid u \in N_{\text{in}}(v)\}\!\}\right)$, where h is an injective function and $c_0 = \mu$. Note, that this variant of WL makes use of edge features and works on directed graphs. While traditionally WL is defined for undirected and unlabeled graphs, this is a common assumption in similar lines of work.

The *unfolding tree* with height i of a node $v \in V(G)$ is defined as the in-tree $F_i^v = (v, \{\!\{F_{i-1}^u \mid u \in N_{\text{in}}(v)\}\!\})$, where $F_0^v = (\{v\}, \emptyset)$. The unfolding trees F_i^v and F_i^w of two nodes v and w are isomorphic if and only if the colors of the nodes in iteration i are equal, see, e.g., Kriege (2022). The Weisfeiler-Leman algorithm has historically been used as a heuristic for graph isomorphism. Let $\mathrm{WL}(G) = \{\!\{c_\infty(v) \mid v \in V(G)\}\!\}$ be the multiset of node colors in the stable coloring (Arvind et al., 2015). Two graphs G and H are not isomorphic if $\mathrm{WL}(G) \neq \mathrm{WL}(H)$. However, non-isomorphic graphs G and H with $\mathrm{WL}(G) = \mathrm{WL}(H)$ exist. WL for example cannot distinguish the molecular graphs in Figure 1 or a 6-cycle from two triangles.

Expressivity. Let $\mathcal{G}$ denote the set of all graphs and $\mathcal{G}_n = \{G \in \mathcal{G} \mid |V(G)| \leq n\}$ for all $n \in \mathbb{N}$. Let ϕ, ψ be two graph embedding algorithms, which map graphs to embedding spaces (e.g., $\mathbb{R}^d$). We say ϕ *is at least as expressive as* ψ if for all pairs of graphs G, G' with $\psi(G) \neq \psi(G')$ it holds that $\phi(G) \neq \phi(G')$. Let $\mathcal{G}' \subseteq \mathcal{G}$ be a family of graphs, for example, the set of all outerplanar graphs. We say that a graph embedding algorithm ϕ is *maximally expressive* for $\mathcal{G}'$ if for every non-isomorphic pair of graphs $G, G' \in \mathcal{G}'$ it holds that $\phi(G) \neq \phi(G')$. We can generalize this to parameterized graph embeddings such as GNNs: Let ϕ_w be a graph embedding with parameters w (e.g., the weights of a neural network). A parameterized graph embedding ϕ_w is maximally expressive for $\mathcal{G}'$ if for all $n \in \mathbb{N}$ there exists a parameter choice w_n such that for every non-isomorphic pair of graphs $G, G' \in \mathcal{G}' \cap \mathcal{G}_n$ it holds that $\phi_{w_n}(G) \neq \phi_{w_n}(G')$.

Hamiltonian adjacency lists. A Hamiltonian cycle of a graph is a cycle containing each node exactly once. A Hamiltonian cycle $(v_1, \ldots, v_k)$ on an undirected graph, can be separated into two directed Hamiltonian cycles $C = (v_1, \ldots, v_k)$ with corresponding edges $v_1v_2, \ldots, v_kv_1$ and $\overleftarrow{C} = (v_k, \ldots, v_1)$ with corresponding edges $v_kv_{k-1}, \ldots, v_1v_k$. Biconnected outerplanar graphs have a unique Hamiltonian cycle that can be found in linear time (Mitchell, 1979). Hamiltonian adjacency lists (HALs) are derived by annotating each node with the sorted distances d_C to all its neighbors on the two directed variants of the Hamiltonian cycle C. Figure 2 shows a graph annotated with its HALs in both directions of the Hamiltonian cycle. Following the Hamiltonian cycle in one direction and concatenating the HALs gives a HAL sequence S (and a reverse sequence R, for the other direction).

A sequence S of length n is a *cyclic shift* of another sequence S' of length n if there exists an $\ell \in \mathbb{N}$ such that $S_i = S'_j$ for all $i \in \{1, \ldots, n\}$ where $j = i + \ell \mod n$. The HAL sequence uniquely identifies a biconnected outerplanar graph (if both directions and cyclic shifts are considered):

Lemma 1 (Colbourn & Booth 1981)**.** *Two biconnected outerplanar graphs G and H with HAL and reverse sequences S_G, S_H and R_G, R_H are isomorphic, iff S_G is a cyclic shift of S_H or R_H.*

Published in Transactions on Machine Learning Research (12/2024)

Table 1: Common benchmark datasets and percentage of outerplanar graphs in them.

Dataset	#Graphs	Outerplanar
ZINC	12000	98 %
PCQM-Contact	529434	98 %
MOLESOL	1128	97 %
MOLTOXCAST	8576	96 %
MOLTOX21	7831	96 %
MOLLIPO	4200	96 %
MOLCLINTOX	1477	94 %
NCI-2000	250251	94 %
peptides-func	15535	93 %
MOLBACE	1513	93 %
MOLSIDER	1427	92 %
MOLBBBP	2039	92 %
MOLHIV	41127	92 %

Table 2: Pre-processing time of CAT on the training splits of all datasets and relative additional training/evaluation time with CAT.

Dataset	CAT Runtime	Train+Eval. Time Change
MOLESOL	2 ± 1 s	26 %
MOLBBBP	5 ± 1 s	36 %
MOLSIDER	6 ± 1 s	21 %
MOLBACE	6 ± 1 s	42 %
MOLLIPO	14 ± 1 s	38 %
MOLTOX21	15 ± 1 s	27 %
MOLTOXCAST	16 ± 1 s	13 %
ZINC	44 ± 1 s	27 %
MOLHIV	152 ± 1 s	31 %

4 Identifying Outerplanar Graphs Using Weisfeiler-Leman

We develop a graph transformation called *cyclic adjacency transform* (CAT), that enables WL to distinguish all outerplanar graphs. We first introduce CAT*, enabling WL to distinguish any pair of non-isomorphic biconnected outerplanar graphs, and then extend it to all outerplanar graphs.

In CAT* (see Section 4.1), nodes are duplicated to represent the Hamiltonian cycle in both directions. We annotate edges not in the Hamiltonian cycle with the distance their endpoints have on the Hamiltonian cycle. This allows the Weisfeiler-Leman algorithm to encode HAL sequences in the unfolding trees of the nodes and in turn distinguish pairs of non-isomorphic biconnected outerplanar graphs.

To extend our transformation to all outerplanar graphs (see Section 4.2), we need to ensure that the biconnected components keep their unique encoding. It should also be ensured that non-isomorphic graphs with the same biconnected components (but arranged or rotated differently) can be distinguished. We address this by introducing articulation and block pooling vertices. These nodes contain all the information about their respective blocks and their position in these blocks. Together with nodes outside of blocks, this forms a forest which encodes the entire original graph. As it is known that WL is maximally expressive on forests, it follows that WL is maximally expressive on such graphs.

4.1 Identifying Biconnected Outerplanar Graphs Using Weisfeiler-Leman

We first present a graph transformation called CAT*, that allows the Weisfeiler-Leman algorithm to distinguish any two non-isomorphic *biconnected* outerplanar graphs. Figure 3 shows an example of CAT*. Note that CAT$^*(G)$ consists of two disjoint copies of G, with directed and annotated edges. We first describe the transformation informally for ease of understanding, followed by the formal definition.

We start with an empty graph and add the (unique) Hamiltonian cycle of the original graph twice, directed in opposite ways (steps 1 and 2). Then, we add the remaining edges of the original graph to both cycles, directed in both directions each (step 3). Finally, we set node and edge features to the original features, and extend edge features with the distance of their endpoints in the Hamiltonian cycle (step 4).

Definition 1. *The* CAT$^*(G) = G'$ *transformation maps a biconnected outerplanar graph* $G = (V, E, \mu, \nu)$ *to a new graph* $G' = (V', E', \mu', \nu')$ *as follows:*

1. *Let* $(v_1, \dots, v_n)$ *be the Hamiltonian cycle of* G *and* C *and* $\overleftarrow{C}$ *be its directed versions.*
2. *Add node and edge disjoint copies of* C *and* $\overleftarrow{C}$ *together with the corresponding edges to an empty graph* G' *and* $\forall e \in E'$ *set* $\nu'(e) = (1, \nu(e))$.

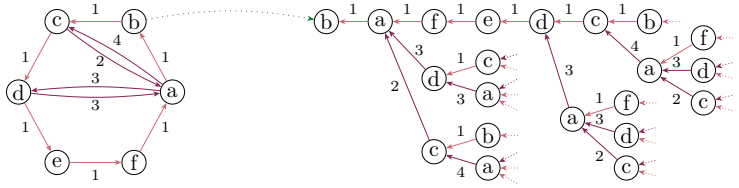


Figure 4: One part of the CAT* transformation of the graph from Figure 3 and an example unfolding tree of one of its nodes from which the HAL sequence of the original graph can be reconstructed.

3. *Let $D \subseteq E$ be the edges of G* not *on the (undirected) Hamiltonian cycle. Add edges in both directions to G' for the copies of C and $\overleftarrow{C}$ for each edge in D: $E' = E' \cup E_d \cup E_{\overleftarrow{d}}$ with $E_d = \bigcup_{v_i v_j \in D} \{v'_i v'_j, v'_j v'_i\}$ and $E_{\overleftarrow{d}} = \bigcup_{v_i v_j \in D} \{v''_i v''_j, v''_j v''_i\}$ for copies v'_i of v_i in C (resp. v''_i in $\overleftarrow{C}$).*
4. *$\forall v'_i, v''_i \in V'$ set $\mu'(v'_i) = \mu'(v''_i) = \mu(v_i)$. $\forall v_i v_j \in E_d$ set $\nu'(v_i v_j) = (d_C(v_j, v_i), \nu(v_i v_j))$ and $\forall v'_i v'_j \in E_{\overleftarrow{d}}$ set $\nu'(v'_i v'_j) = (d_{\overleftarrow{C}}(v'_j, v'_i), \nu(v_j v_i))$.*

Using CAT* we prove our first main result.

Theorem 1. *Two biconnected outerplanar graphs G and H are isomorphic, if and only if* $\mathrm{WL}(\mathrm{CAT}^*(G)) = \mathrm{WL}(\mathrm{CAT}^*(H))$.

Proof. Two graphs are distinguished by WL if and only if the multisets of node colors of their stable colorings differ. Trivially, $|V(G)| \neq |V(H)| \Rightarrow |V(\mathrm{CAT}^*(G))| \neq |V(\mathrm{CAT}^*(H))| \Rightarrow \mathrm{WL}(\mathrm{CAT}^*(G)) \neq \mathrm{WL}(\mathrm{CAT}^*(H))$, so we only focus on graphs with $|V(G)| = |V(H)|$. Two nodes only get the same color if their unfolding trees are isomorphic. The first number in the HAL of each node is always 1, so it can be ignored, and the last number is always $|V(G)| - 1$, so this can simply be reconstructed by $|V(\mathrm{CAT}^*(G))|$. The rest of the HAL sequence and the node labels of G can be reconstructed from the unfolding tree of any node in $\mathrm{CAT}^*(G)$: Trivially, each node has two direct neighbors in the Hamiltonian cycle. In the unfolding tree these are the parent and the single child with the 1-annotated edge. All other neighbors in the HAL can be reconstructed by looking at the weights of the edges that do not have weight 1. Figure 4 shows an example. Looking at any two biconnected outerplanar graphs with n nodes, the Weisfeiler-Leman algorithm will be able to distinguish them after at most n iterations, iff they are non-isomorphic: Since the HAL sequence is encoded in the unfolding trees from all starting points (cyclic shift) and in both directions (reverse direction), this identifies isomorphism by Lemma 1. □

4.2 Extending CAT* to All Outerplanar Graphs

While CAT* is defined for single blocks (biconnected outerplanar components), an outerplanar graph can contain multiple of them, as well as additional connections. We define the CAT transformation by applying CAT* to the blocks of the graphs and adding additional nodes and edges, which enable WL to distinguish any two non-isomorphic outerplanar graphs. We first give an informal description to provide an intuition for the various steps before stating the formal definition.

Starting from an empty graph, add the graph induced by all edges not in blocks and nodes in more than one block (steps 1 and 2). For each block, add the result of CAT* applied to the block (step 3.1). We refer to nodes created in this step as Hamiltonian cycle nodes. Add pooling nodes connecting the two Hamiltonian cycle nodes corresponding to the same original node (step 3.2). Add a node for each block (block nodes) and connect it to the pooling nodes of that block (step 3.3). Relabel the nodes from step 2, that belong to at least one block (articulation nodes) and connect them to the pooling nodes corresponding to the same original node (step 3.6). Finally, create a global (block) pooling node and connect it to the block nodes (step 4). For each node we add, its initial features (if present) are extended by a label referring to the type (Hamiltonian cycle node, pooling node, articulation node, etc.).

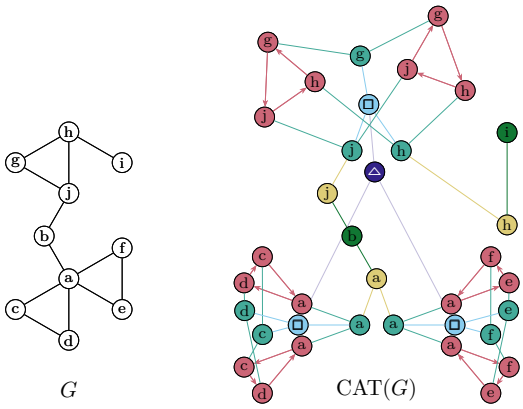


Figure 5: A graph and its CAT transformation. Original node labels are represented by letters, edge and node labels from the CAT transform are represented by colors. Note that CAT(G) looks like a cat.

Definition 2. *The* CAT$(G) = G'$ *transformation maps a graph* G *to a new graph* G' *as follows:*

1. *Let* $B_1, \ldots, B_\ell$ *be the blocks of* G *and let* F *be the graph induced by the edges of* G *that are not in any block plus the nodes that are present in more than one block. Let* $\{\perp, \square, \bowtie, \star, \triangle\}$ *be distinct node labels not in* X*.*
2. *Add* F *to* G' *with labels* $\mu'(v) = (\perp, \mu(v))$ *for all* $v \in F$*.*
3. *For each block* B_i *in* G*:*
   - 3.1. *Let* $B_i', \overleftarrow{B_i'}$ *be the two connected components in* CAT$^*(B_i)$*. Add* B_i' *and* $\overleftarrow{B_i'}$ *to* G'*.*
   - 3.2. *For all pairs* $(v, \overleftarrow{v})$ *of corresponding nodes in* B_i' *and* $\overleftarrow{B_i'}$ *add a node* p_v *with* $\mu'(p_v) = (\star, \mu(v))$ *and edges* $p_v v, v p_v, p_v \overleftarrow{v}, \overleftarrow{v} p_v$ *to* G'*.*
   - 3.3. *Add a node* b_i *to* G' *with* $\mu'(b_i) = \square$*. For all* $v \in V(B_i)$ *add edges* $b_i p_v$ *and* $p_v b_i$*.*
   - 3.4. *Let* $A_i = V(B_i) \cap V(F)$ *be the nodes of* B_i *in* F*.*
   - 3.5. *Let* $\gamma_i : A_i \to V(B_i')$ *map nodes of* F *to their copy in* B_i'*.*
   - 3.6. *For each* $a \in A_i$*, let* $\mu'(a) = (\bowtie, \mu(a))$ *and add edges* $p_{\gamma_i(a)} a$ *and* $a p_{\gamma_i(a)}$ *to* G'*.*
4. *Add node* g *with* $\mu'(g) = \triangle$ *to* G' *and for all nodes* b_i*, add edges* $g b_i$ *and* $b_i g$ *to* G'*.*
5. *Let* CAT$(G) = G'$*.*

An example of the CAT transformation can be seen in Figure 5. Appendix F contains additional visualizations of the transformation on real-life molecular graphs. Informally, a graph is transformed using CAT by uniquely encoding all its blocks (using CAT*) and then connecting the results to encode also the position within the graph and the orientation using the original graph structure. All nodes and edges that are not part of a block are of course also added to the transformed graph.

We can now prove our main result, which implies that MPNNs together with CAT are maximally expressive on outerplanar graphs.

Theorem 2. *Two outerplanar graphs* G *and* H *are isomorphic, if and only if* WL(CAT(G)) $=$ WL(CAT(H))*.*

Proof. Following Theorem 1, each block will be uniquely identified by WL. Since the additional nodes have distinct labels, they will not cause WL to falsely report two blocks as isomorphic when they are not. The

Table 3: Resistance and diameter before and after the CAT transformation. $\overline{\rho(G)}$ and $\max_{\rho(G)}$ denote the average and maximum pair-wise effective resistance for graph G. Results are reported as mean and standard deviation across all graphs in the datasets. In all cases, smaller is better.

Dataset	$\Phi(G)$	$\Phi(\text{CAT}(G))$	$\overline{\rho(G)}$	$\overline{\rho(\text{CAT}(G))}$	$\max_{\rho(G)}$	$\max_{\rho(\text{CAT}(G))}$
ZINC	12.5 ± 2.6	$\mathbf{9.9 \pm 1.6}$	4.0 ± 0.7	$\mathbf{2.6 \pm 0.4}$	10.0 ± 2.0	$\mathbf{7.7 \pm 1.9}$
MOLESOL	$\mathbf{6.6 \pm 3.3}$	6.9 ± 3.8	2.3 ± 1.0	$\mathbf{2.1 \pm 0.9}$	$\mathbf{5.5 \pm 2.3}$	6.0 ± 2.8
MOLTOXCAST	8.5 ± 4.7	$\mathbf{8.4 \pm 4.0}$	3.0 ± 1.5	$\mathbf{2.6 \pm 1.3}$	$\mathbf{7.2 \pm 4.3}$	7.4 ± 3.9
MOLTOX21	8.8 ± 4.6	$\mathbf{8.7 \pm 4.0}$	3.1 ± 1.5	$\mathbf{2.7 \pm 1.3}$	$\mathbf{7.5 \pm 4.2}$	7.7 ± 3.9
MOLLIPO	13.8 ± 4.0	$\mathbf{9.9 \pm 2.1}$	4.3 ± 1.2	$\mathbf{2.6 \pm 0.5}$	10.7 ± 3.4	$\mathbf{7.9 \pm 2.3}$
MOLBACE	15.1 ± 3.2	$\mathbf{11.5 \pm 2.8}$	5.0 ± 1.3	$\mathbf{2.9 \pm 0.7}$	12.5 ± 3.4	$\mathbf{9.1 \pm 2.6}$
MOLSIDER	12.6 ± 11.8	$\mathbf{10.4 \pm 7.3}$	4.1 ± 3.8	$\mathbf{2.9 \pm 2.2}$	10.4 ± 11.0	$\mathbf{8.9 \pm 6.8}$
MOLBBBP	10.7 ± 3.7	$\mathbf{9.1 \pm 2.6}$	3.4 ± 1.1	$\mathbf{2.4 \pm 0.6}$	8.3 ± 3.8	$\mathbf{7.5 \pm 2.5}$
MOLHIV	11.9 ± 5.2	$\mathbf{9.9 \pm 3.8}$	3.9 ± 1.7	$\mathbf{2.7 \pm 1.2}$	9.3 ± 4.7	$\mathbf{8.2 \pm 3.8}$

information about the entire HAL sequence of each block is stored in the block nodes b. The pooling nodes p connect the block and block nodes to the rest of the graph (through the articulation nodes a), determining the orientation of the block. Note that the graph returned by CAT without the CAT* blocks and the global pooling node g is a tree. Relying on the labels of the pooling and block nodes, we can reconstruct the original graph from this tree. As WL can distinguish non-isomorphic labeled trees (Arvind et al., 2015; Kiefer, 2020), it can thus distinguish non-isomorphic outerplanar graphs using CAT. For the other direction, note that CAT is permutation-invariant: for two isomorphic graphs G and H, the graphs $\text{CAT}(G)$ and $\text{CAT}(H)$ are isomorphic and WL will give the same coloring for both. □

Importantly, we can compute $\text{CAT}(G)$ in linear time. The computational complexity is dominated by the computation of the blocks (Tarjan, 1972) and their Hamiltonian cycles (Mitchell, 1979), which both require linear time. Note that we only add a linear number of nodes and edges. From Morris et al. (2019) and Xu et al. (2019) it follows, that MPNNs, that are as expressive as 1-WL, can distinguish $\text{CAT}(G)$ and $\text{CAT}(H)$ for non-isomorphic outerplanar graphs G and H.

Note that our proof used an important property of the WL algorithm: Adding uniquely labeled nodes and edges to WL-distinguishable graphs never leads to WL-indistinguishable graphs. We use this property to add a global pooling node in step 4 of CAT which is connected to all block pooling nodes. This allows to pass messages between block nodes in fewer iterations in the subsequent MPNN step.

CAT can also be applied to non-outerplanar graphs. In this case, our graph transformation performs the steps described in Definition 2. However, if a non-outerplanar biconnected component B_i is encountered, only one copy B_i' is created in $\text{CAT}(G)$ and its vertices are connected to the corresponding pooling nodes. An example for this is depicted in Appendix D. While this never reduces expressivity, it is also not guaranteed to improve expressivity on non-outerplanar graphs. Note that it can be determined in linear time whether a block is outerplanar while trying to compute the Hamiltonian cycle of the block (Mitchell, 1979). Hence, the CAT transformation always only requires linear time.

4.3 Influence of CAT on Graph Connectivity

As poor graph connectivity may negatively influence the predictive performance of GNNs (Alon & Yahav, 2021), we investigate the effects of CAT on different measures of graph connectivity such as the diameter $\Phi(G)$ of a graph. We refer to the shortest path distance between two nodes as the *distance* between them.

Observation 1. *For a block B of a graph G, it holds that $\Phi(\text{CAT}(B)) \leq 4$.*

Proof. Let $u, v \in V(\text{CAT}(B))$. By definition all nodes in $\text{CAT}(B)$ are either from a Hamiltonian cycle created by CAT*, a pooling node, or a block node. If both nodes are from a Hamiltonian cycle, then there is a path u, p_u, b, p_v, v between them, where p_u, p_v are pooling nodes and b is the block node. Hence, $d_{\text{CAT}(G)}(u, v) \leq 4$. If u or v is a pooling or a block node, then the above path implies that $d_{\text{CAT}(G)}(u, v) < 4$. □

Table 4: Predictive performance of MPNNs with and without CAT on different molecular benchmark datasets. Arrows indicate whether smaller (↓) or bigger (↑) results are better. **Bold** entries are an MPNN with CAT that outperforms the same MPNN without CAT.

Dataset → ↓ Model	ZINC MAE ↓	ZINC250k MAE ↓	MOLHIV ROC-AUC ↑	MOLBACE ROC-AUC ↑	MOLBBBP ROC-AUC ↑
GIN	0.168 ± 0.007	0.033 ± 0.003	77.9 ± 1.0	74.6 ± 3.2	66.0 ± 2.1
CAT+GIN	$\mathbf{0.101 \pm 0.004}$	0.034 ± 0.003	76.7 ± 1.8	$\mathbf{79.5 \pm 2.5}$	$\mathbf{67.2 \pm 1.8}$
GCN	0.184 ± 0.013	0.067 ± 0.005	76.7 ± 1.4	77.9 ± 1.7	66.1 ± 2.4
CAT+GCN	$\mathbf{0.123 \pm 0.008}$	$\mathbf{0.034 \pm 0.003}$	$\mathbf{77.1 \pm 1.6}$	$\mathbf{79.2 \pm 1.5}$	$\mathbf{68.3 \pm 1.7}$
GAT	0.375 ± 0.013	0.103 ± 0.004	76.6 ± 2.0	81.7 ± 2.3	66.2 ± 1.4
CAT+GAT	$\mathbf{0.201 \pm 0.022}$	$\mathbf{0.046 \pm 0.004}$	75.3 ± 1.6	79.3 ± 1.6	66.0 ± 1.9
Dataset → ↓ Model	MOLSIDER ROC-AUC ↑	MOLESOL RMSE ↓	MOLTOXCAST ROC-AUC ↑	MOLLIPO RMSE ↓	MOLTOX21 ROC-AUC ↑
GIN	56.6 ± 1.0	1.105 ± 0.077	65.3 ± 0.6	0.717 ± 0.016	75.8 ± 0.7
CAT+GIN	$\mathbf{58.2 \pm 0.9}$	$\mathbf{0.985 \pm 0.055}$	$\mathbf{65.6 \pm 0.5}$	0.798 ± 0.031	74.8 ± 1.0
GCN	56.7 ± 1.5	1.053 ± 0.087	64.4 ± 0.4	0.748 ± 0.018	76.4 ± 0.3
CAT+GCN	$\mathbf{57.9 \pm 1.8}$	$\mathbf{1.006 \pm 0.036}$	$\mathbf{66.2 \pm 0.8}$	0.771 ± 0.023	74.9 ± 0.8
GAT	58.4 ± 1.0	1.037 ± 0.063	63.8 ± 0.8	0.728 ± 0.024	76.3 ± 0.6
CAT+GAT	58.3 ± 1.3	1.09 ± 0.048	$\mathbf{64.5 \pm 0.8}$	0.754 ± 0.021	75.4 ± 0.7

Observation 2. *Let B_i and B_j be two blocks of a graph G. In $\mathrm{CAT}(G)$, the maximum distance between any node in $\mathrm{CAT}(B_i)$ and any node in $\mathrm{CAT}(B_j)$ is 6.*

Proof. Let $u \in V(\mathrm{CAT}(B_i))$ and $v \in V(\mathrm{CAT}(B_j))$. If $B_i = B_j$, then Observation 1 implies $d_{\mathrm{CAT}(G)}(u, v) \leq 4$. If $B_i \neq B_j$, then there exists a path $u, p_u, b_i, g, b_j, p_v, v$ where p_u, p_v are pooling nodes, b_i, b_j are the block node for block B_i, B_j, and g is the global block pooling node. Thus, $d_{\mathrm{CAT}(G)}(u, v) \leq 6$. □

Observations 1 and 2 show that for some subgraphs CAT provides constant upper bounds in diameter. On the graph level, CAT can only lead to a small increase of the diameter.

Proposition 1. *For an outerplanar graph G, $\Phi(\mathrm{CAT}(G)) \leq \Phi(G) + 7$.*

Proof sketch. To prove the proposition we bound the distance between any pair of nodes in $\mathrm{CAT}(G)$ by case analysis based on the type of nodes. We defer the full proof to Appendix B. □

In most practical cases, the short-cutting inside or between blocks leads to CAT reducing the graph diameter (see Observations 1 and 2). In Table 3, we demonstrate this on molecular benchmark datasets. Besides the diameter, another useful graph connectivity measure is the *effective resistance*. The notion of effective resistance originates in electrical engineering (Kirchhoff, 1847) and has implications on several graph properties. For example, the effective resistance between two nodes is proportional to the commute time between them (Chandra et al., 1989). Intuitively, a large effective resistance between two nodes suggests that information propagation between the nodes is hindered. Recently, effective resistance has been in fact linked to *over-squashing* (Black et al., 2023) in GNNs, which is a negative effect that leads to long-range interactions having little impact on the predictions of a GNN. Effective resistance as introduced by Kirchhoff (1847) is naturally only defined for undirected graphs. As CAT produces directed graphs, we therefore use an extension of effective resistance introduced by Young et al. (2015) that is applicable to directed graphs. We refer to Young et al. (2015) for more details. In Table 3 we demonstrate that CAT reduces the pair-wise effective resistance on molecular benchmark datasets.

5 Experimental Evaluation

We investigate whether our proposed method CAT can improve the predictive performance of MPNNs on molecular benchmark datasets.[1] We utilize three commonly used MPNNs: GIN (Xu et al., 2019), GCN (Kipf & Welling, 2017), and GAT-v2 (Veličković et al., 2018; Brody et al., 2022). We train on the commonly used `ZINC` (Gómez-Bombarelli et al., 2018; Sterling & Irwin, 2015) and `MOLHIV` (Hu et al., 2020) datasets, which contain graphs representing molecules. We supplement these with 7 small datasets (see Table 4) from the OGB collection (Hu et al., 2020). In addition to the `ZINC` dataset with 12k graphs we also use the larger `ZINC250k` variant with 250k graphs. In total, we train three MPNNs on ten datasets with and without CAT. For each configuration, we separately tune hyperparameters on the validation set and train a model with the best hyperparameters ten times on the training set and evaluate it on the test set. The only exception to this is `ZINC250k` where we evaluate the final hyperparameter configuration five times, due to the large dataset size. For each dataset we report the mean and standard deviation of the most common evaluation metric on the test set in the epoch with the best validation performance. For `ZINC` we use a batch size of 128 and an initial learning rate of 10^{-3} that gets halved if the validation metric does not improve for 20 epochs. The training stops after 500 epochs or if the learning rate dips below 10^{-5}. For all other datasets we train with a fixed learning rate for 100 epochs and a batch size of 128. We use the same hyperparameter grid for all models and provide more details in Appendix E. Besides measuring the predictive performance, we also measure the time needed for applying CAT (averaged over ten runs), and the training and evaluation time for GIN and GIN+CAT with the same hyperparameters on all datasets (averaged over five runs). Finally, we report the values for the diameters and effective resistances as described in Section 4.3.

Results. Table 2 shows the pre-processing time of CAT. Note that this is the performance of running CAT on only a single CPU core. It is possible to achieve faster runtimes by simply parallelizing different graphs over different cores. This negligible runtime allows applying the transformation even in realistic high-throughput screening applications (Krasoulis et al., 2022), for example, on `MOLHIV` it takes only 5ms to process a graph. Training and prediction time on CAT transformed graphs increases by 29% on average. Table 4 shows the predictive performance of all models. Note that our baseline models obtain strong results, often surpassing the performance of (higher-order) GNNs reported in the literature, and that we train each MPNN and MPNN+CAT with exactly the same sets of hyperparameters. Overall, CAT improves the predictive performance of GIN and GCN in the majority of datasets (6 / 10 and 8 / 10, respectively). For GIN and GCN, performance increases reliably on all datasets, except `MOLLIPO` and `MOLTOX21`. Surprisingly, CAT does not work well with GAT and only improves its performance in 2 / 10 datasets. Most notably on `ZINC`, CAT achieves very strong results boosting the predictive performance of MPNNs between 33% (GCN) and 46% (GAT). This is only surpassed by higher-order GNNs such as CW Networks (Bodnar et al., 2021a) which obtain a MAE of 0.079 ± 0.006 at the cost of potentially exponential pre-processing runtime due to enumerating cycles in the graph. Table 3 shows that CAT reduces both graph diameter and maximum pair-wise resistance on most datasets. Furthermore, CAT reduces the average pair-wise resistance in all datasets. This suggests that CAT is effective at improving graph connectivity in real-life molecular graphs.

6 Conclusion

We proposed *Cyclic Adjacency Transform* (CAT), a linear time graph transformation, that enables the Weisfeiler-Leman algorithm to be maximally expressive on outerplanar graphs. We rely on the fact that biconnected outerplanar graphs can be uniquely identified by their Hamiltonian adjacency list sequences, which CAT encodes in unfolding trees. It follows that a combination of MPNNs and CAT can distinguish all outerplanar graphs. We achieved promising empirical results on standard molecular benchmark datasets where CAT improved the performance of GIN and GCN in most cases, while for GAT we could not observe this benefit. Computing CAT takes linear time and our implementation of CAT typically runs in the order of seconds. Motivated by the recent interest in the over-squashing phenomenon, we also studied the effect of CAT on graph connectivity. We proved that in the worst-case CAT increases the diameter of outerplanar graphs by a small additive constant. However, inspecting CAT on real-world data, we find that

[1] Our code can be found at https://github.com/ocatias/outerplanarGNN.

the diameter decreases most of the time. Similarly, we observed that the maximum and average pair-wise effective resistance, which is associated with over-squashing, typically decreases after applying CAT.

Our work highlights the value of GNNs designed for specific graph classes. This perspective aligns with a recent position paper, in which Morris et al. (2024) propose to investigate the runtime complexity of maximally expressive GNNs on practically relevant graphs (Challenge II.4). In general, achieving high expressivity is computationally expensive. However, for some graph classes this might not be the case. In this work, we have demonstrated that for *outerplanar* graphs maximal expressivity can be achieved in linear time.

Focusing on specific graph classes allows making use of existing graph theoretical results. This could lead to novel GNNs that are both runtime efficient and highly expressive. Possible applications are infrastructure and road networks, which are characterized by *near-planar* graphs that tend to have low *graph degeneracy* (Eppstein & Gupta, 2017). More generally, it seems promising to study sparse graphs, such as *bounded expansion* graphs (Nešetřil & De Mendez, 2012). Finally, in the future we plan to extend our work to k-outerplanar graphs, which are known to capture even more molecular graphs (Horváth et al., 2010).

Acknowledgement

This work was supported in part by the Vienna Science and Technology Fund (WWTF) through projects [10.47379/VRG19009] and [10.47379/ICT22059]; by the TU Wien DK SecInt; and by the Austrian Science Fund (FWF) through project NanOX-ML (6728). MT acknowledges support from a DOC fellowship of the Austrian academy of sciences (ÖAW).

References

Uri Alon and Eran Yahav. On the bottleneck of graph neural networks and its practical implications. In *ICLR*, 2021.

Vikraman Arvind, Johannes Köbler, Gaurav Rattan, and Oleg Verbitsky. On the power of color refinement. In *Fundamentals of Computation Theory*, 2015.

Beatrice Bevilacqua, Fabrizio Frasca, Derek Lim, Balasubramaniam Srinivasan, Chen Cai, Gopinath Balamurugan, Michael Bronstein, and Haggai Maron. Equivariant subgraph aggregation networks. In *ICLR*, 2021.

Lukas Biewald. Experiment tracking with weights and biases, 2020. URL https://www.wandb.com/. Software available from wandb.com.

Mitchell Black, Zhengchao Wan, Amir Nayyeri, and Yusu Wang. Understanding oversquashing in GNNs through the lens of effective resistance. In *ICML*, 2023.

Hans L. Bodlaender. A partial k-arboretum of graphs with bounded treewidth. *Theoretical Computer Science*, 209(1-2):1–45, 1998.

Cristian Bodnar, Fabrizio Frasca, Nina Otter, Yu Guang Wang, Pietro Liò, Guido Montúfar, and Michael Bronstein. Weisfeiler and Lehman go cellular: CW networks. In *NeurIPS*, 2021a.

Cristian Bodnar, Fabrizio Frasca, Yu Guang Wang, Nina Otter, Guido Montufar, Pietro Lio, and Michael Bronstein. Weisfeiler and Lehman go topological: Message passing simplicial networks. In *ICML*, 2021b.

Giorgos Bouritsas, Fabrizio Frasca, Stefanos Zafeiriou, and Michael M Bronstein. Improving graph neural network expressivity via subgraph isomorphism counting. *Transactions on Pattern Analysis and Machine Intelligence*, 45(1):657–668, 2022.

Shaked Brody, Uri Alon, and Eran Yahav. How attentive are graph attention networks? In *ICLR*, 2022.

Ashok K Chandra, Prabhakar Raghavan, Walter L Ruzzo, and Roman Smolensky. The electrical resistance of a graph captures its commute and cover times. In *STOC*, 1989.

Charles J. Colbourn and Kellogg S. Booth. Linear time automorphism algorithms for trees, interval graphs, and planar graphs. *SIAM Journal on Computing*, 10(1):203–225, 1981.

Leonardo Cotta, Christopher Morris, and Bruno Ribeiro. Reconstruction for powerful graph representations. *NeurIPS*, 2021.

Reinhard Diestel. *Graph theory*. Springer, 2024.

Radoslav Dimitrov, Zeyang Zhao, Ralph Abboud, and Ismail Ceylan. Plane: Representation learning over planar graphs. *NeurIPS*, 2023.

Andre Droschinsky, Nils Kriege, and Petra Mutzel. Finding largest common substructures of molecules in quadratic time. In *International Conference on Current Trends in Theory and Practice of Informatics*, 2017.

David Eppstein and Siddharth Gupta. Crossing patterns in nonplanar road networks. In *Proceedings of the 25th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems*, 2017.

István Fáry. On straight line representation of planar graphs. *Acta Sci. Math.(Szeged)*, 11:229–233, 1948.

Stefan Felsner. *Geometric graphs and arrangements: some chapters from combinatorial geometry*. Springer, 2012.

Matthias Fey and Jan E. Lenssen. Fast graph representation learning with PyTorch Geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.

Fabrizio Frasca, Beatrice Bevilacqua, Michael M. Bronstein, and Haggai Maron. Understanding and extending subgraph GNNs by rethinking their symmetries. In *NeurIPS*, 2022.

Rafael Gómez-Bombarelli, Jennifer N. Wei, David Duvenaud, José Miguel Hernández-Lobato, Benjamín Sánchez-Lengeling, Dennis Sheberla, Jorge Aguilera-Iparraguirre, Timothy D. Hirzel, Ryan P. Adams, and Alán Aspuru-Guzik. Automatic chemical design using a data-driven continuous representation of molecules. *ACS Central Science*, 2018.

Tamás Horváth and Jan Ramon. Efficient frequent connected subgraph mining in graphs of bounded tree-width. *Theoretical Computer Science*, 411(31-33):2784–2797, 2010.

Tamás Horváth, Jan Ramon, and Stefan Wrobel. Frequent subgraph mining in outerplanar graphs. In *KDD*, 2006.

Tamás Horváth, Jan Ramon, and Stefan Wrobel. Frequent subgraph mining in outerplanar graphs. *Data Mining and Knowledge Discovery*, 21(3):472–508, 2010.

Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. Open Graph Benchmark: Datasets for machine learning on graphs. In *NeurIPS*, 2020.

Neil Immerman and Eric Lander. *Describing graphs: A first-order approach to graph canonization*. Yale University Press, 1990.

Sandra Kiefer. *Power and limits of the Weisfeiler-Leman algorithm*. PhD thesis, RWTH Aachen University, 2020.

Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. In *ICLR*, 2017.

Gustav Kirchhoff. Ueber die Auflösung der Gleichungen, auf welche man bei der Untersuchung der linearen Vertheilung galvanischer Ströme geführt wird. *Annalen der Physik*, 148(12):497–508, 1847.

Agamemnon Krasoulis, Nick Antonopoulos, Vassilis Pitsikalis, and Stavros Theodorakis. Denvis: Scalable and high-throughput virtual screening using graph neural networks with atomic and surface protein pocket features. *Journal of Chemical Information and Modeling*, 62(19):4642–4659, 2022.

Nils M Kriege. Weisfeiler and Leman go walking: Random walk kernels revisited. In *NeurIPS*, 2022.

Haggai Maron, Heli Ben-Hamu, Hadar Serviansky, and Yaron Lipman. Provably powerful graph networks. *NeurIPS*, 2019.

Sandra L. Mitchell. Linear algorithms to recognize outerplanar and maximal outerplanar graphs. *Information Processing Letters*, 9(5):229–232, 1979.

Christopher Morris, Martin Ritzert, Matthias Fey, William L Hamilton, Jan Eric Lenssen, Gaurav Rattan, and Martin Grohe. Weisfeiler and Leman go neural: Higher-order graph neural networks. In *AAAI*, 2019.

Christopher Morris, Fabrizio Frasca, Nadav Dym, Haggai Maron, Ismail Ilkan Ceylan, Ron Levie, Derek Lim, Michael M Bronstein, Martin Grohe, and Stefanie Jegelka. Position: Future directions in the theory of graph machine learning. In *ICML*, 2024.

Jaroslav Nešetřil and Patrice Ossona De Mendez. *Sparsity: graphs, structures, and algorithms*, volume 28. Springer Science & Business Media, 2012.

Leander Schietgat, Jan Ramon, and Maurice Bruynooghe. A polynomial-time maximum common subgraph algorithm for outerplanar graphs and its application to chemoinformatics. *Annals of Mathematics and Artificial Intelligence*, 69(4):343–376, 2013.

Teague Sterling and John J. Irwin. Zinc 15 – ligand discovery for everyone. *Journal of Chemical Information and Modeling*, 2015.

Maciej M. Sysło. The subgraph isomorphism problem for outerplanar graphs. *Theoretical Computer Science*, 17(1):91–97, 1982.

Robert Endre Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal of Computing*, 1(2): 146–160, 1972.

Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph Attention Networks. In *ICLR*, 2018.

Zhenqin Wu, Bharath Ramsundar, Evan N. Feinberg, Joseph Gomes, Caleb Geniesse, Aneesh S. Pappu, Karl Leswing, and Vijay Pande. Moleculenet: a benchmark for molecular machine learning. 2018.

Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks? In *ICLR*, 2019.

George Forrest Young, Luca Scardovi, and Naomi Ehrich Leonard. A new notion of effective resistance for directed graphs—part I: Definition and properties. *Transactions on Automatic Control*, 61(7):1727–1736, 2015.

Bohang Zhang, Guhao Feng, Yiheng Du, Di He, and Liwei Wang. A complete expressiveness hierarchy for subgraph GNNs via subgraph Weisfeiler-Lehman tests. In *ICML*, 2023a.

Bohang Zhang, Shengjie Luo, Liwei Wang, and Di He. Rethinking the expressive power of GNNs via graph biconnectivity. In *ICLR*, 2023b.

Published in Transactions on Machine Learning Research (12/2024)

A Outerplanar Graphs

Let $G = (V, E)$ be an undirected graph. A *planar embedding* of G consists of an injective mapping f of nodes V to vectors in the plane $\mathbb{R}^2$ and for each pair of adjacent nodes u, v a continuous path between $f(u)$ and $f(v)$ such that no pair of such paths cross. More formally, a continuous path from $f(u)$ to $f(v)$ is a continuous function $p : [0, 1] \to \mathbb{R}^2$ such that $p(0) = f(u)$ and $p(1) = f(v)$. Two such continuous paths p, p' cross if there exist $t, t' \in (0, 1)$ such that $p(t) = p'(t')$. The graph G is *planar* if it has a planar embedding. By Fáry's theorem each planar graph always has a planar embedding with straight line segments as paths (Fáry, 1948). Intuitively a planar embedding dissects the plane into regions called *faces*. Formally these are the (topologically) connected regions of $\mathbb{R}^2$ with all the edge paths p removed. Any planar embedding has a unique face not bounded by paths (the only non-compact face), which is called the *outer face*. A graph is *outerplanar* if it has a planar embedding with all nodes lying on the boundary of the outer face. See, for example, Felsner (2012) for more details.

We can also characterize outerplanar graphs using forbidden *graph minors*. A graph H is a minor of G, if H can be obtained from G by a series of node deletion, edge deletion, and edge contractions (i.e., removing an edge and replacing the two endpoints by a new node). Let $K_{a,b}$ be the *complete bipartite graph* with bi-partition $A \cup B = V$ with $a = |A|$ and $b = |B|$ and K_c be the *complete* graph on c nodes. A graph G is outerplanar if and only if G has no $K_{2,3}$ minor and no K_4 minor. Similarly, planar graphs can be characterized by the Kuratowski theorem as graphs with no $K_{3,3}$ and no K_5 minor. For more details see, for example Diestel (2024).

B Proof of Proposition 1

We prove Proposition 1 which states that $\Phi(\mathrm{CAT}(G)) \leq \Phi(G) + 7$ for every outerplanar graph G.

Proof. Let $u, v \in V(\mathrm{CAT}(G))$ such that $d_{\mathrm{CAT}(G)}(u, v) = \Phi(\mathrm{CAT}(G))$. We call a node a *tree node* if it was not part of a block in G and was not created by CAT. A node that is *not* a tree node is either a Hamiltonian cycle node, a pooling node, a block pooling node, or a global block pooling node.

Case 1: Both u, v are not tree nodes. By Observation 2: $d_{\mathrm{CAT}(G)}(u, v) \leq 6$.

Case 2: Node u is a tree node and v is not. Let $a \in V(G)$ be the closest articulation node to u in $\mathrm{CAT}(G)$. Then, there is a path of length $d_G(u, a)$ in $\mathrm{CAT}(G)$ from u to a. We can extend this path by one node to reach a pooling node. By Definition 2, there exists a path of length at most 6 from this pooling node to v. Thus $d_{\mathrm{CAT}(G)}(u, v) \leq d_G(u, a) + 7 \leq \Phi(G) + 7$.

Case 3: Both u, v are tree nodes.

Case 3a: Suppose that the shortest path between u and v in G does not contain any edge inside of an outerplanar block, then $d_{\mathrm{CAT}(G)}(u, v) = d_G(u, v) \leq \Phi(G)$.

Case 3b: Suppose that the shortest path between u and v in G contains one or more edges inside exactly one block. Then, we can enter and exit this block in $\mathrm{CAT}(G)$ through a path a_1, p_1, b, p_2, a_2, where a_1, a_2 are articulation nodes, p_1, p_2 are pooling nodes, and b is a block node. Note that the articulation nodes were part of the path in G which implies $d_{\mathrm{CAT}(G)}(u, v) = d_G(u, a_1) + d_G(a_2, v) + 4 = d_G(u, v) - d_G(a_1, a_2) + 4$. Furthermore, we do not need to take the one or more edges inside the block to go from a_1 to a_2. Using $d_G(a_1, a_2) \geq 1$ we obtain $d_{\mathrm{CAT}(G)}(u, v) = d_G(u, v) - d_G(a_1, a_2) + 4 \leq d_G(u, v) + 3 \leq \Phi(G) + 3$.

Case 3c: Suppose that the shortest path between u and v in G contains two or more edges that are contained in two or more different blocks. Then, for $\mathrm{CAT}(G)$ we can shortcut from the first to the last block node of the shortest path between u and v through a path $a_1, p_1, b_1, g, b_2, p_2, a_2$, where a_1, a_2 are articulation nodes, p_1, p_2 are pooling nodes, b_1, b_2 are block nodes, and g is the global block pooling node. Note that the articulation nodes were part of the path in G which implies that $d_{\mathrm{CAT}(G)}(u, v) = d_G(u, a_1) + d_G(a_2, v) + 6 = d_G(u, v) - d_G(a_1, a_2) + 6$. By assumption, we know that $d_G(a_1, a_2) \geq 2$ which implies $d_{\mathrm{CAT}(G)}(u, v) = d_G(u, v) - d_G(a_1, a_2) + 6 \leq d_G(u, v) + 4 \leq \Phi(G) + 4$. □

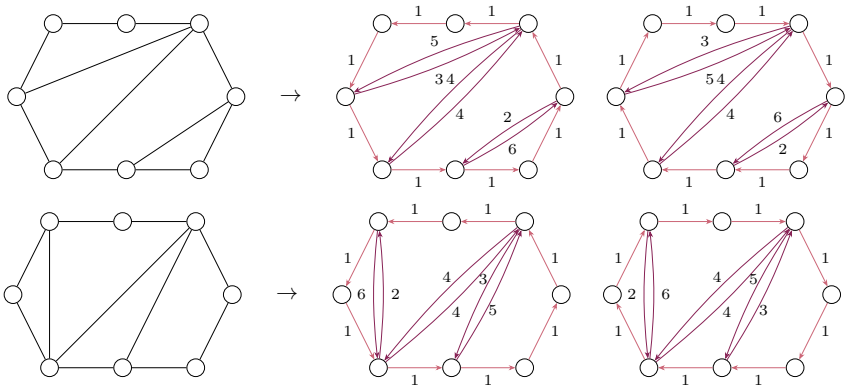


Figure 6: Additional examples for CAT*. The biconnected outerplanar graphs on the left are transformed using Definition 1.

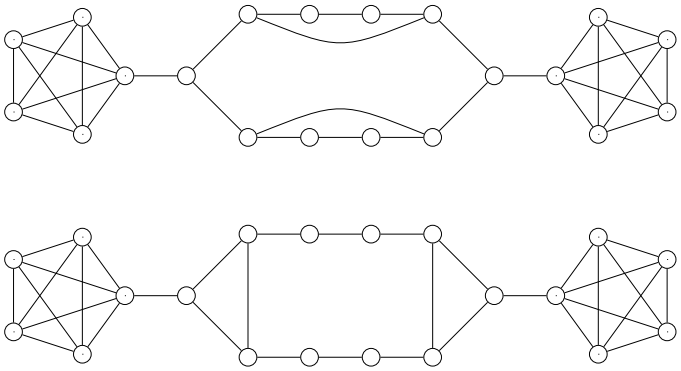


Figure 7: Two nonplanar graphs which PlanE cannot distinguish. However, CAT can distinguish the graphs by their two central outerplanar biconnected components.

C Additional Examples for CAT*

Here, we provide additional examples for CAT*. Figure 6 shows two graphs and the results, when transformed using CAT*. As described in Definition 1, the original graph is copied twice and the edges of the Hamiltonian cycle are directed in each direction once (with their label extended with 1). The remaining edges are added to each copy directed in both directions, with their label being extended to describe the distance of their two nodes when only using edges from the Hamiltonian cycle.

D Incomparable Expressivity of CAT and PlanE

The expressivity of CAT and PlanE is incomparable. This follows from Propositions 2 and 3, showing that there are pairs of graphs that are distinguishable by CAT and not PlanE and vice versa.

Proposition 2. *There exists non-planar graphs that* CAT *can distinguish and PlanE cannot distinguish.*

Proof. The claim follows from Figure 7. □

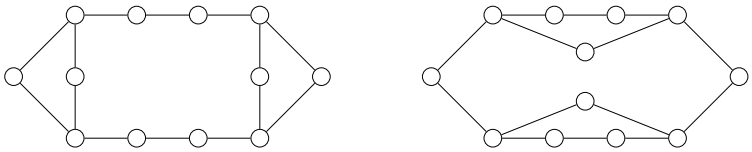


Figure 8: Two planar (but not outerplanar) graphs which CAT cannot distinguish.

Proposition 3. *There exists planar graphs that PlanE can distinguish and* CAT *cannot distinguish.*

Proof. The claim follows from Figure 8. □

E Details for Experimental Evaluation

Our models are implemented in PyTorch-Geometric (Fey & Lenssen, 2019) and trained on a single NVIDIA GeForce RTX 3080 GPU. We use WandB (Biewald, 2020) for tracking. The used server has 64 GB of RAM and an 11th Gen Intel(R) Core(TM) i9-11900KF CPU running at 3.50GHz. Table 5 shows the hyperparameters of our MPNNs for different datasets. We use the same hyperparameter grid for MPNNs combined with CAT. We used a smaller hyperparameter grid for `MOLHIV` as `MOLHIV` is larger than the most of the other datasets, meaning that training takes much longer. When benchmarking the speed of GIN against GIN+CAT we train for 100 epochs with a batch size of 128 on all datasets with the same hyperparameters for both models (see Table 5).

CAT **implementation.** CAT adds an additional feature to each node which encodes the type of that node, i.e., nodes from Hamiltonian cycles, block nodes, pooling nodes, articulation nodes and or global block nodes. Furthermore, we create additional edge features encoding the types of nodes incident to this edge i.e., an edge between two different nodes in a Hamiltonian cycle has a different type than an edge from a pooling node to the block node. For newly created nodes and edges we set their remaining features to the feature of the node / edge they are based on; for example, a pooling node will have the features of the node they are performing the pooling operation for. For nodes that have no natural representation in the graph (block and block pooling nodes) we set these features to 0. To ensure that only these nodes get assigned 0 features, we shift the values of these features for all other nodes by 1.[2] Note that our MPNNs treat the distance on edges in blocks as a categorical feature. Representing the distances as numerical features did not improve performance in preliminary experiments.

Table 5: Hyperparameter grids for GIN, GCN and GAT on different datasets.

Parameter	All datasets except `MOLHIV`	`MOLHIV`	Speed Benchmarks (Table 2)
Message passing layers	2, 3, 4, 5	4, 5	4
Final MLP layers	2	2	2
Pooling operation	mean, sum	mean, sum	mean
Embedding dimension	64, 128, 256	64,128	64
Jumping knowledge	last	concat	concat
Dropout rate	0, 0.5	0.5	0

Dataset description. All graphs in our datasets represent molecules and have graph-level tasks. For all tasks we use the metrics that are commonly used by the community, e.g., MAE for `ZINC` or ROC-AUC

[2]This assumes that the features are categorical and not numerical–which is the case for the used datasets.

for MOLHIV. For ZINC (10k graphs) and ZINC250k (250k graphs) (Gómez-Bombarelli et al., 2018; Sterling & Irwin, 2015) the task is to predict the constrained solubility which is a regression task. All other datasets are from the Open Graph Benchmark (Hu et al., 2020) based on MoleculeNet (Wu et al., 2018) and the task is always to predict some molecular property. On MOLHIV, the task is to predict whether a molecule inhibits HIV virus replication or not (Hu et al., 2020). For the tasks of the other datasets, we refer to Wu et al. (2018).

F Additional Figures

We provide additional visualizations of the CAT transformation. In all figures, the color of the vertices in the transformed graph have the following meaning: red nodes are from Hamiltonian cycles, blue nodes correspond to blocks, yellow nodes pool the nodes from Hamiltonian cycles, orange nodes correspond to articulation nodes and the gray node pools block nodes. Figure 9 shows a synthetic example with a non-outerplanar graph. Figure 10 demonstrates CAT on various synthetic graphs. Figure 11 shows the result of CAT on molecular graphs from ZINC and MOLHIV. Somewhat ironically, CAT often generates frog graphs on MOLHIV as can be seen in Figure 12.

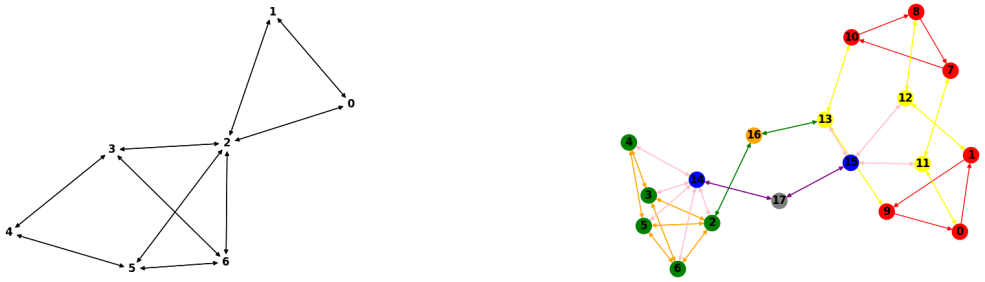


Figure 9: Left: example non-outerplanar graph. Right: result of applying CAT to the graph. Colors indicate the type of node (see Appendix F).

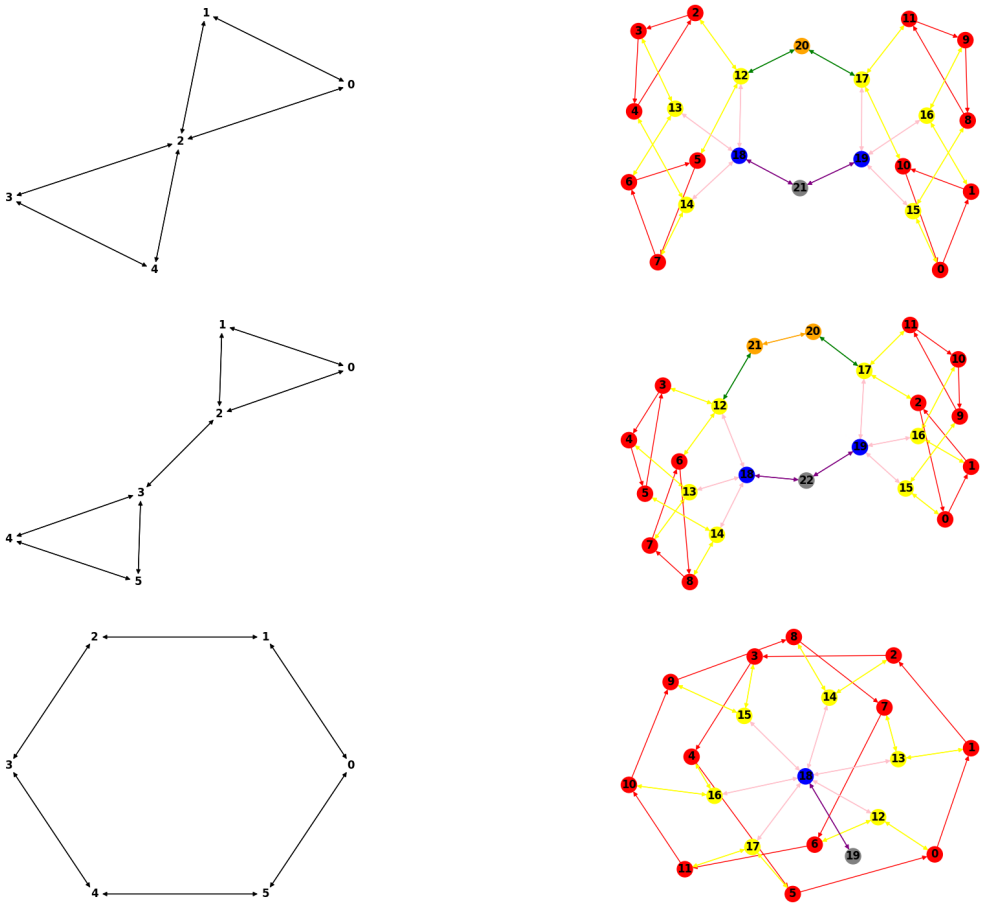


Figure 10: Left: example graphs. Right: result of applying CAT to these graphs. Colors indicate the type of node (see Appendix F).

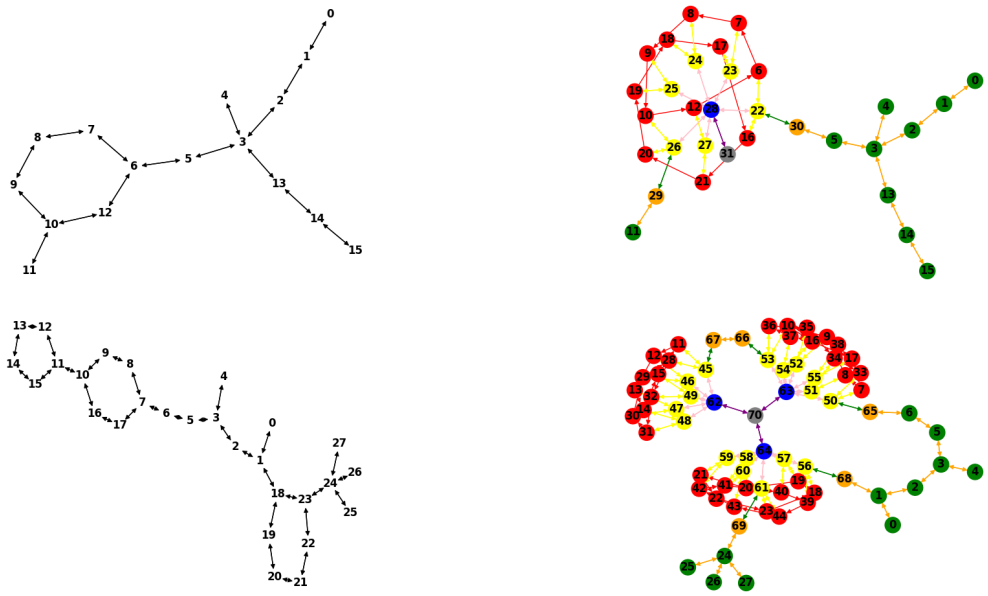


Figure 11: Left: example graphs from `MOLHIV` (top) and `ZINC` (bottom). Right: result of applying CAT to these graphs. Colors indicate the type of node (see Appendix F).

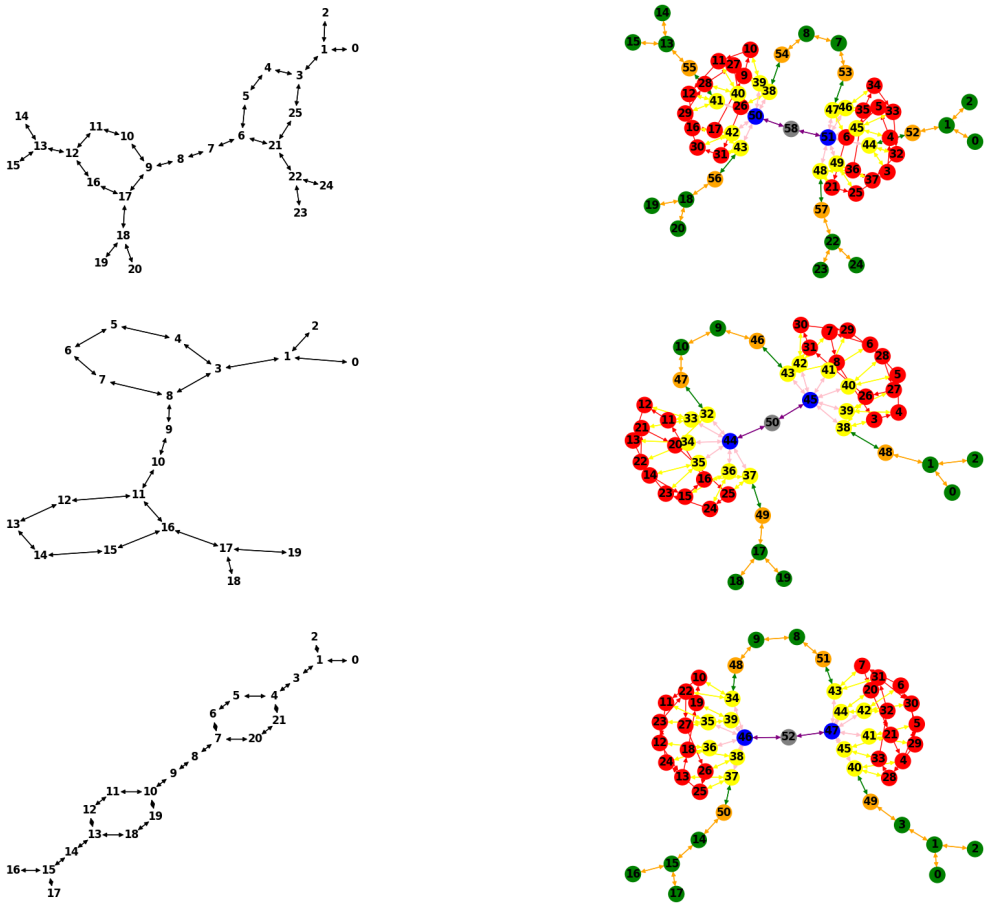


Figure 12: Left: example graphs from `MOLHIV`. Right: result of applying CAT to the graph. Colors indicate the type of node (see Appendix F).