Università della Calabria
Department of Mathematics and Computer Science

Master's Degree Course in
Artificial Intelligence and Computer Science

Master's Thesis

Approximating Graph Edit Distance through Graph Neural Networks: Methods, Limitations, and Proposals

Supervisors:
Ch.mo. Giorgio Terracina
Dott. Sebastiano Antonio Piccolo

Candidate:
Federico Calabrò
Matricola 247976

Academic Year 2023/2024

Contents

1 Introduction

Graphs are basic data structures in computer science and mathematics which are used to represent relationships between elements. It has vertices (or nodes) and edges (or links) which are the connections between any two vertices. This simple yet rich notation can model many realistic cases and therefore is a useful tool for studying many systems. For instance, social networks can be modeled as graphs where nodes are the people, and edges are the relationships between the people, which allows analyzing the social processes, the diffusion of information, and the formation of communities. All these can be analyzed using graphs in biology such as protein-protein interaction networks, neural networks of the brain and ecological networks. In the same way, in transportation, the cities can be the nodes and the roads or the flights can be the edges in a network that enables finding the best way in a city, planning of cities, and the best use of resources in terms of logistics. They can also be used to describe communication networks whereby the devices are represented as nodes and the connection as edges to help determine the flow of data, the strength of the network and the best way to allocate resources.

The structures of these networks can be described by graph theory and different characteristics like connectivity, centrality, and clustering coefficient can be used to analyze the characteristics of the structures. Connectivity is the extent of the nodes' connection, centrality determines the most influential nodes within the graph, and the clustering coefficient provides an understanding on the likelihood of the nodes to cluster. Other important properties of a graph include; graph diameter, which is defined as the maximum of the shortest path between any two nodes, and graph density which gives an idea of how connected the nodes in the graph are. These properties assist in the discovery of important information about the structure and behavior of the graph and, therefore, the analysis and decision-making process.

The Graph Edit Distance (GED) problem is a key aspect in the graph theory since it gives a measure of graph similarity. GED measures the number of operations needed to transform one graph into another for example, insertions, deletions and substitutions of nodes and edges. This measure is extremely useful for a number of purposes, for example in bioinformatics where it can be used to compare the shapes of molecules in order to find new drugs or study the evolution. In computer vision, GED is important in object recognition where the structural distance between the graphical models of different objects needs to be compared in order to differentiate between them. Other graph similarity measures include graph isomorphism which compares the structural similarity of two graphs exactly and subgraph isomorphism which determines whether one graph is a sub graph of another and is used in pattern recognition and in searching for nodes and their relationships in large networks.

Hence when the actual GED between two graphs is well understood it can be very meaningful. For instance, in the field of bioinformatics, the identification of the relationship between different molecular conformations can make it possible to find new drugs and target new treatments, by identifying structural features that are associated with biological activity. In the context of SNA, GED is useful in identifying nodes that form communities or clusters in a network where nodes with similar patterns of connections are grouped together and may be used in the identification of key nodes, spread of information or formation of social groups. In addition, in the application of pattern recognition and image analysis, GED can be adopted to determine the objects and relations of the structures of these objects to improve the precision and authenticity of the automatic systems. The measure of graph similarity makes it easier and more effective to compare the graphs in these areas, which in turn leads to the development of new and better solutions.

However, finding the exact GED is a challenging task because of the reason that it is computationally expensive. In fact, the problem is NP-hard, and thus the time needed to find the solution increases exponentially with the size of the graphs, and becomes infeasible for large graphs. This is a brute force method, where the search for the most optimal edit path is carried out on all the possible paths and this is not feasible in real life situations. Several heuristics and approximation algorithms have been suggested, yet they suffer from the problem of achieving a reasonable compromise between the quality of the solution produced and the time taken to produce it, which affects the credibility of the results. NP-hard problems are those that are at least as difficult as the best problems in NP, and for which no efficient, fast solutions are known. This inherent difficulty serves to enhance the problem of computing GED and, therefore, the need to come up with better approximation techniques that can give good results within a short time.

Neural networks, which are the basis of the current machine learning, are the advanced tools that are used for effective processing of numerous multi-dimensional data. A neural network is a set of models for solving a particular problem in a way that is reminiscent of the brain's structure. It comprises a number of layers of nodes or neurons which are connected and which can take in information and produce outputs. Neural networks are fed with a lot of data which they use to fine tune the value of the various parameters that are present in their structure based on the error between the predicted output and the actual output. This training method includes forward propagation that entails feeding input data through the network to derive output and back propagation where the error is taken through the network to adjust weights in a bid to enhance precision of the model. Neural networks have been applied to a variety of problems with high success rates in image and speech recognition, natural language processing and more recently graph data analysis which has shown the flexibility of the approach.

Graph Neural Networks (GNNs), are a class of Neural Networks which are designed to operate on graph data type. These architectures, called GNNs, try to take advantage of the graph structure by doing convolutions over the nodes and edges of the graph, as well as local and global properties of the graph. This makes them suitable for a number of applications such as node classification, link prediction and graph classification. Because GNNs can learn the complex patterns and representations, they can be applied for approximating the GED.

GNNs work by enhancing the node representation at each step, with respect to its neighbors, thus capturing the relations and interdependencies in the graph. This is because the iterative process is useful in the learning of hierarchical representations that are essential in the understanding and analysis of graph-structured data, and thus improves the accuracy of the predictions in various applications.

To this end, this thesis surveys the state-of-the-art methods in GED computation, including the neural network-based methods. All the works under review propose different approaches to handling the challenges of GED computation, and everyone is innovative in its way. Through the critical comparison of these methods this review seeks to determine the effectiveness as well as the weakness and opportunities that may be harnessed to enhance their effectiveness. The paper that launched the work on SimGNN [3] is critical to the field, offering a strong foundation to work on. Some of the most recent works like GedGNN [12] try to go further and offer new ideas and enhance earlier approaches.

Optimization of the GED computation is very crucial especially for applications that depend on graph similarity measures. For instance, improved and more accurate GED can be used to accelerate and enhance the comparison of molecular geometries which in turn may help in the identification of new therapeutic agents. In social network analysis, it can help in the identification of more precise communities, which as a result can help in understanding the social processes and can be useful in controlling the same, thus making the interventions and policies more effective. In computer vision, the advanced techniques can improve the object recognition systems and make them more accurate and fast, which is useful in several fields such as auto-mobiles to surveillance systems. The importance of better GED computation is seen in many fields, which shows that there is still much work to be done in this area to discover new applications and improvements.

In the course of this review of these articles, the aim is to present an overview of the state of the art in GED computation. Thus, focusing on the best practices and identifying directions for future studies, this thesis is intended to help develop the existing approaches to GED computation. A replication of the results of important recent works such as the work that presented GedGNN will also be performed. Furthermore, this thesis will provide constructive feedback on some issues like the quality of code and the fairness of the results, as well as the drawbacks of the datasets employed. A talk about problems such as the poor quality of the dataset and recommendation of solutions will also be given: artificial dataset creation, and the creation of neural networks that can be tested on any dataset, which will make the evaluation more equitable. This paper offers a systematic review of the methods in an attempt to identify the existing gaps and the possibilities that may lead to the development of new and better methods for use in graph theory and all its applications. It is the hope of this thesis to offer a detailed discussion and critical appraisal in order to inform future research and development work; to point the way to further improvements that will guarantee the effectiveness and applicability of GED computation methods across different disciplines.

2 Graph Data Structure

A *graph* G [Figure 1] is a nonlinear data structure consisting of a set of vertices and arcs, where arcs connect pairs of vertices in the set. Graphs are widely used to represent relationships between entities and play a significant role in the development of fields like Computer Science, Optimization, Chemistry and others. They are a pillar in network-based systems modeling such as social media, biological networks, and transportation systems, being a crucial tool for analyzing and solving complex problems. Use cases of graphs can be found in the actual world, for example in recommendation systems, routing and navigation algorithms like GPS, optimization problems and resource allocation (also known as transportation problems).

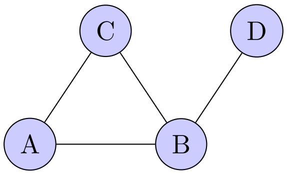


Figure 1: Simple undirected graph example.

A graph can be formally defined as a tuple $G = (V, E)$, where:

- V is a finite set of vertices where each represents an entity or a data point. The set V is often denoted as $V = \{v_1, v_2, \ldots, v_n\}$ where n is the number of vertices.

- E is a set of edges, where each edge is an unordered pair of distinct vertices from V. Thus, $E \subseteq \{\{u, v\} \mid u, v \in V \text{ and } u \neq v\}$. Edges represents the existing relationship between two vertices in the set.

For instance, graph depicted in Figure 1 can be formally defined as a tuple $G = (V, E)$, where:

- V is the set of vertices, $V = \{A, B, C, D\}$

- E is the set of edges, $E = \{(A, B), (A, C), (B, C), (B, D)\}$

2.1 Types of Graphs

There exist different categories of graphs depending on their properties, including:

- **Directed Graph** [Figure 2]: also known as digraph, is the case where the direction is indicated on the edges, representing G as an ordered pair (u, v) where $u, v \in V$ and $u \neq v$. It's applied in a range of areas, including web page ranking, where links between pages have a set direction, and citation networks, where one paper references another.

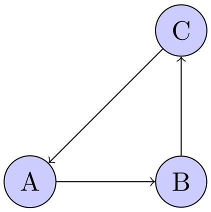


Figure 2: Directed graph where edges have directions indicated by arrows.

- **Undirected Graph** [Figure 3]: the edges do not have a direction, represented as an unordered pair $\{u, v\}$ where $u, v \in V$ and $u \neq v$. This kind of graph is commonly used to model networks where the connections of two nodes are mutual, indicating that relationship is valid in both senses.

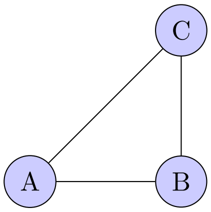


Figure 3: Undirected graph example where edges are bidirectional (there are no arrows).

- **Weighted Graph** [Figure 4]: in this graph edges have a weight (or cost) related, represented as a function $w : E \to \mathbb{R}$ where $w(e)$ is the weight of edge $e \in E$. This is particularly useful in transportation networks where the weights can express distances, the time spent traveling from one point to another, or costs associated with the displacement.

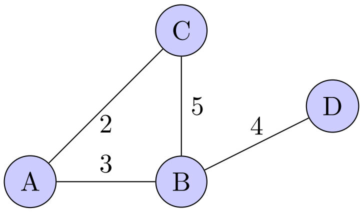


Figure 4: Weighted graph example where each edge is labeled with a weight.

- **Simple Graph** [Figure 5]: is a graph without loops (there doesn't exist a path from a vertex to itself) and has no multiple edges (the same pair of vertices is not connected more than once). Simple graphs are the most basic type of graph existing, with straightforward structures that make them easy to handle. They are often used for modeling basic networks to maintain a clear design and facilitate the analysis of the structure and the understanding of network properties.

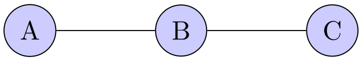


Figure 5: Simple graph example with a linear connection between vertices A, B, and C, with no loops or multiple edges.

- **Complete Graph** [Figure 6]: is a graph in which every vertex is connected with all the other vertices in the set. Formally, a complete graph on n vertices, denoted as K_n, has $E = \{\{u, v\} \mid u, v \in V, u \neq v\}$. They are widely used in cases where is necessary maximum connectivity, such as some network topologies and combinatorial optimization problems.

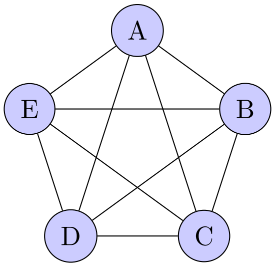


Figure 6: Complete graph example where each node is connected to each other.

- **Bipartite Graph** [Figure 7]: a graph whose vertices are separable into two disjoint sets U and W in a way that an edge only connects a vertex from U with a vertex from W. Bipartite graphs are useful for modeling relationships between objects from two different classes. For example, in the context of job assignment, vertices in U can symbolize jobs and vertices in W workers. Edges will indicate which job is assigned to each worker.

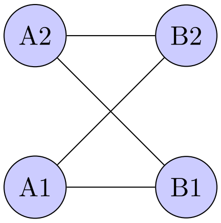


Figure 7: Bipartite graph example with two distinct sets of vertices with edges connecting vertices across the sets but not within them.

- **Multigraph** [Figure 8]: it is a graph where multiple edges occur between the same pair of vertices. Formally, $G = (V, E)$ where E is a multiset of unordered pairs of vertices. Multigraphs are often used for modeling networks where multiple relationships or interactions exist for the same pair of vertices. For example, it is known that in transportation networks exists multiple routes or connections between two locations.

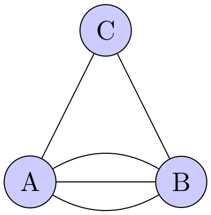


Figure 8: Multigraph example with multiple edges between vertices A and B.

- **Cyclic Graph** [Figure 9]: is the one that contains at least a cycle, being a cycle a path where every vertex on it is reachable from itself. Cyclic graphs are employed to model processes or systems where feedback loops are present. For example, it could be said that in certain biological systems or recurrent neural networks, loops represent the actions or the recurrent connections respectively.

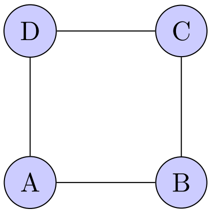


Figure 9: Cyclic graph example with a cycle A-B-C-D-A.

- **Acyclic Graph** [Figure 10]: An acyclic graph is a graph without loops. A direct acyclic graph (DAG) is a directed graph without loops. Acyclic graphs, specially DAGs, are mainly used for modeling cases as task scheduling, where dependencies should not form cycles. In this kind of situation, a task will start only if all its prerequisites are completed. The absence of loops ensures that there aren't circular dependencies.

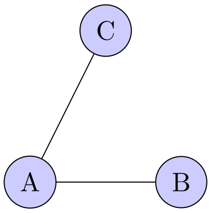
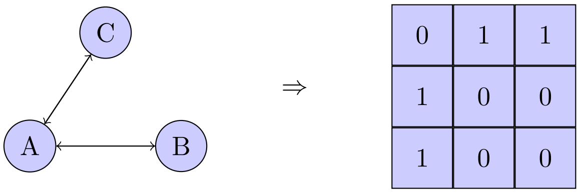


Figure 10: Acyclic graph example with no cycles.

2.2 Graph Representation

There are several manners to represent a graph, including:

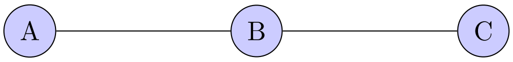
- **Adjacency Matrix** [Figure 11]: An adjacency matrix A corresponding to a graph $G = (V, E)$ is a binary square matrix of size $|V| \times |V|$ that expresses the existence of a relationship between a pair of vertices. The value of A_{ij} is 1 if there is an edge connecting vertices v_i and v_j, and 0 otherwise. This structure is particularly convenient for dense graphs where the number of edges is nearest to the limit of possible edges. It facilitates efficiency in the querying process to know if an edge exists and is easy to implement for algorithms that require constant monitoring of the edge's presence. Even so, the space complexity is $O(|V|^2)$, which is a problem for large graphs with many vertices.



0	1	1
1	0	0
1	0	0

Figure 11: Adjacency Matrix example.

- **Adjacency List** [Figure 12]: An adjacency list is a collection of lists where each one corresponds to a vertex and contains all the adjacent vertices. Having a graph $G = (V, E)$, the adjacency list could be implemented as a list array $\{L_1, L_2, \ldots, L_n\}$ and every L_i will include all v_i's neighbors. These structures are more efficient for sparse graphs due to the number of edges is much smaller than the number of possible edges. The use of an adjacency list facilitates the work of graph algorithms such as breadth-first search (BFS) and depth-first search (DFS), where only the more important neighbors need to be visited.



$$[\ Adj(A) = [B], \quad Adj(B) = [A, C], \quad Adj(C) = [B]\ ]$$

Figure 12: Adjacency List example.

2.3 Properties of Graphs

Graphs have some outstanding properties that help to analyze them and determine the best application for each type, including:

- **Degree** [Figure 13]: The degree of a vertex is the number of vertices in the graph that incident on it. Formally, for a vertex v in a graph $G = (V, E)$, the degree $\deg(v)$ is the number of vertices connected to it. In the case of direct graphs, the in-degree stands for the number of incoming edges, and the out-degree stands for the number of outgoing edges. Vertices with high degrees generally play a crucial role in a graph, indicating significant or highly connected nodes.

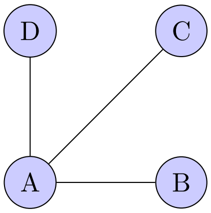


Figure 13: Degree example showing vertex A with degree 3.

- **Connectivity** [Figure 14]: Connectivity refers to the way nodes are connected within the graph. A graph is called connected if there exists a path between every pair of vertices, in other words, each vertex is reachable from any other vertex in the graph. This is a key property for understanding reliability and network robustness and ensures that all nodes can communicate directly or indirectly between them.

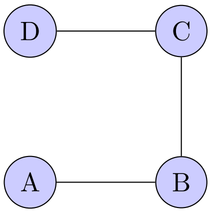


Figure 14: Connectivity example showing a connected graph.

- **Centrality** [Figure 15]: The centrality measures are employed to identify the most significant vertices in a graph. Some of the most important metrics are degree centrality, which quantifies the direct connections to a vertex; closeness centrality, which evaluates the velocity of a node to reach another; and betweenness centrality, which assesses how many times a vertex acts as a bridge in the closeth path between a pair of vertices. These metrics offer distinct points of view about the importance and influence of a node in the net.

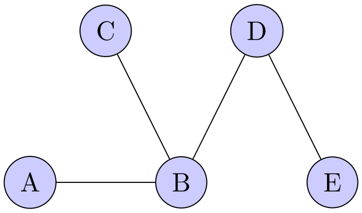


Figure 15: Centrality example showing vertex B as a central node with high degree centrality.

- **Clustering Coefficient** [Figure 16]: The clustering coefficient of a vertex measures how connected the neighbors of that vertex are to each other. A high clustering coefficient suggests a community closely linked in the graph. In mathematics terms, it is defined as the proportion between the real edges and the possible edges among the vertex neighbors.

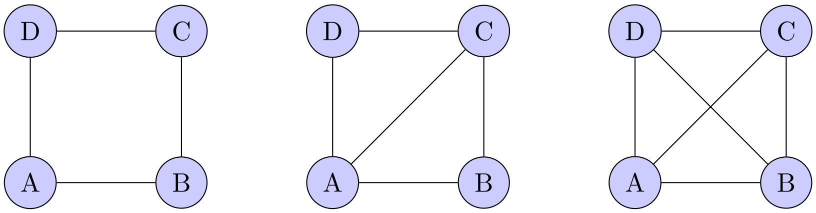


Figure 16: Esempi di coefficienti di clustering: a sinistra un quadrato (coefficient di clustering = 0), al centro un quadrato con diagonale (coefficient di clustering = 5/6), a destra un grafo completo (coefficient di clustering = 1).

- **Graph Diameter** [Figure 17]: The diameter of a graph is the length of the longest shortest paths between any pair of vertices. This metric indicates the "spread" of the graph and helps to understand how distant the vertices are, considering the minimum distance that connects them.

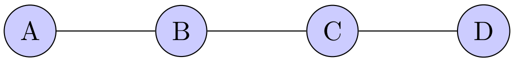


Figure 17: Graph diameter example showing the longest shortest path A-B-C-D with diameter 3.

- **Graph Density** [Figure 18]: The density of a graph is defined as the proportion between the number of existing edges and the maximum possible edges among the vertices. In an undirected graph with n vertices, the total of possible edges is $\frac{n(n-1)}{2}$. The density indicates how close the graph is to completeness, it means, how close it is to having all possible edges.

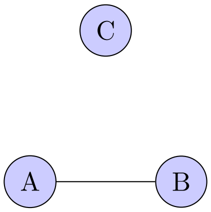


Figure 18: Graph density example showing a sparse graph with few edges relative to the number of vertices.

3 Neural Networks

A *neural network* is a complex computational model inspired by anatomy of the human brain. These models are designed to learn and particularly to recognize patterns in a given data by imitating the functionalities of biological neurons. In fact, the building blocks of every existing neural network are called neurons or nodes. Each of these unit performs simple computations that when combined together allow to tackle a wide range of tasks.

3.1 Basic Structure of a Neural Network

Neurons in a neural networks are organized in *layers* which determine the structure and the capability of the net itself. There is a plenitude of way to organize models, but the simplest one [Figure 19] consists only of three layers.

- **Input Layer**: The first layer of every net. It consists of input neurons that receive the initial data. Usually, each neuron in the input layer corresponds to a feature or example in the input dataset. For instance, if there is used an image recognition model, each neuron might represent a pixel value of the input image.
- **Hidden Layer**: Intermediate layer where the actual computation and learning is performed. In the simplest case, there is just one hidden layer,

but in complex networks there can be many more. Each hidden layer consists of neurons that apply *weights* and *activations functions* to the inputs received from the previous layer.

- **Output Layer**: The last layer in the network, which produces the actual output. For example, when a model is built for a regression task the output layer could be composed of a single neuron which will produce a numeric value as prediction.

The following figure represents a basic neural network with one hidden layer, showing how data flows from the input layer, through the hidden layer, to the output layer:

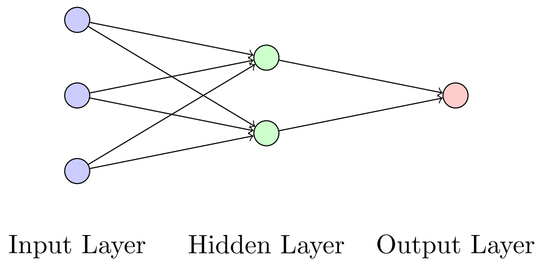


Figure 19: A simple neural network with one hidden layer.

3.1.1 Activation Functions

As mentioned before, in a neural network, each neuron is connected to one or more neurons in the next layer (with exception of the output layer) through *activation functions*. These functions are crucial because they introduce non-linearity into the model, allowing it to capture and learn complex relationships within the data. Without activation functions, it could be built a net with thousands of hidden layers but it would still be limited to ”linear predictions”. Some commonly used activation functions include:

- **Sigmoid**: This function maps any real number into the range (0, 1). It is often used in the output layer for binary classification problems where a probability is needed as output.

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

- **Tanh (Hyperbolic Tangent)**: This function maps any real number into the range (-1, 1). It is zero-centered, which helps in having a more balanced output.

$$\tanh(x) = \frac{2}{1 + e^{-2x}} - 1$$

- **ReLU (Rectified Linear Unit)**: This function is the most commonly used because simple yet effective. It outputs the input directly if it is positive; otherwise, it outputs zero.

$$\text{ReLU}(x) = \max(0, x)$$

3.2 Training Neural Networks

Supervised Learning is a machine learning approach where the data is said to be *labeled*, meaning that for each *example* its *class* is known. There exist Unsupervised and Semi-supervised learning, differents types of automatic learning that will not be discussed in detail as they are not used in this work. Weights are numerical values associated with the connections between neurons, usually being in the range [0, 1] when the data is normalized. Training a neural networks means to find the optimal weights values for each connection so to minimize the error between model's prediction and the actual target values. This is usually achieved through the employment of a technique known as *backpropagation* and the usage of an optimization algorithm such as *descent gradient*. The training process is conducted iteratively until net's performance start to degrade or simply it stops learning. This is done with the usage of a dataset usually split in three parts:

- **Training Set**: This subset is used to adjust the weights of the network when performing the actual training. The model learns and updates its weights based on this data to minimize the error between its predictions and the actual values. Usually it constitutes about the 80% of the whole dataset and if it isn't enough big then techniques to generate artificial data are used.

- **Validation Set**: This small subset is used to tune *hyperparameters*, which are the parameters set before the training process begins. Common hyperparameters include the learning rate, the number of hidden layers, the optimization algorithm, the number of neurons in each layer among others. Hyperparameters thus could influence the model's architecture significantly and hence finding the right values is crucial for achieving optimal performance. Hyperparameters are usually tested within a limited search space and the best ones are then selected. It's especially important to prevent a phenomena known as *overfitting*.

- **Test Set**: This small to medium sized subset is used to evaluate the model's performance on data that it has not seen before. By testing the model on new data, unbiased measures are obtained to evaluate the model in a fair way.

Overfitting occurs when a model learns the training data too well, capturing noise and details. This leads to high accuracy on the training set but poor performance on the test set. To mitigate and prevent this problem, techniques such as regularization, dropout, and early stopping are employed to ensure the model generalizes well to unseen data. The final goal is to have a model that generalize well with respect to any input, and the secret to achieve this is to have good data as first thing.

3.2.1 Backpropagation

Learning for a neural networks means to iteratively apply a forward and a backward pass. In the forward pass, the input data is propagated through the network layer by layer until the output layer is reached. Then the error with respect to the prediction is calculated using a *loss function*, such as mean squared error or mean absolute error. The *gradient* of a function is a fundamental concept in the field of optimization theory because it indicates the direction in which the function increases. This concept is used in the backward pass, where the gradient of the loss function with respect to each weight of the network is calculated by using the technique note as backpropagation [Figure 20] and backpropagated by applying the chain rule. By exploiting this mechanism over and over, weights are adjusted in a manner to minimize the error.

The loss function $L(\mathbf{y}, \hat{\mathbf{y}})$ measures the difference between the predicted output $\hat{\mathbf{y}}$ and the actual output $\mathbf{y}$. For example, in a regression task, the mean squared error (MSE) can be used with the following formule:

$$L(\mathbf{y}, \hat{\mathbf{y}}) = \frac{1}{N} \sum_{i=1}^{N} (y_i - \hat{y}_i)^2$$

Backpropagation uses the chain rule to compute the gradient of the loss function with respect to each weight. The chain rule is a fundamental theorem in calculus used to compute the derivative of the composition of two or more functions. If a variable z depends on y, and y depends on x, then the chain rule states the following:

$$\frac{dz}{dx} = \frac{dz}{dy} \cdot \frac{dy}{dx}$$

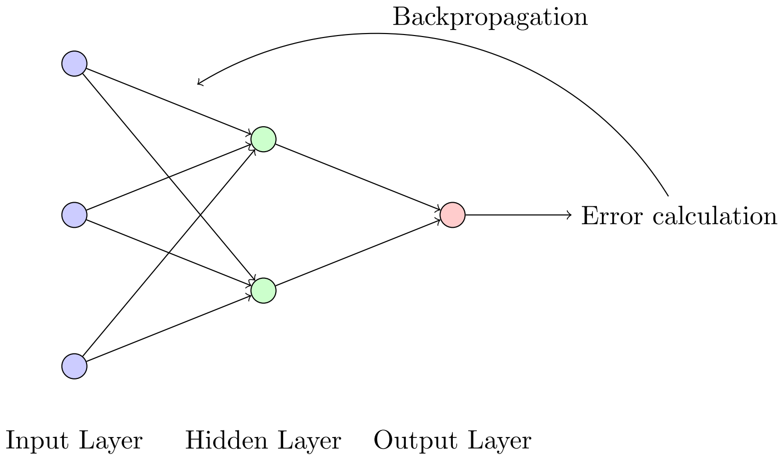


Figure 20: Backpropagation in action within a simple neural network.

3.2.2 Gradient Descent

Once the gradient of the loss function is calculated, an optimization algorithm such as gradient descent [Figure 21] is used to iteratively update the weights by shifting them in the opposite direction of the gradient. How much to move them corresponds to the learning rate hyperparameter and is where an optimization algorithm often differs from another. The learning rate needs to be carefully chosen because it might prevent the finding of a minima thus avoiding the convergence of the model. The weight update rule for a weight w can generally be expressed as:

$$w \leftarrow w - \eta \frac{\partial L}{\partial w}$$

where η is the learning rate. There is a plenitude of optimization algorithms, such as the stochastic gradient descent (SGD), Adam and RMSprop, each offering different trade-offs between computation time and convergence stability. It is worth saying that many modern and more complex methods also are capable of dynamically adjusting the learning rate value during the training process.

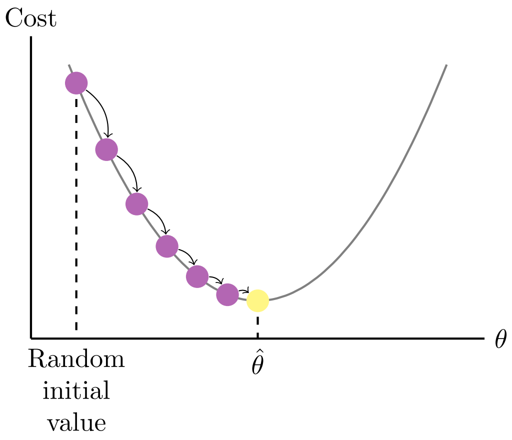


Figure 21: Gradient descent method applied on a concave function.

3.3 Advanced Topics in Neural Networks

3.3.1 Deep Neural Networks

Deep neural networks (DNN) [Figure 22] differ from simple ones for having multiple hidden layers between the input and output layers. The increased depth allows DNNs to model data of higher order of complexity with respect to simple nets, often allowing for better performances. Each (hidden) layer in a DNN can be thought as learning at a different level of abstraction, with the early layers capturing low-level features and deeper layers capturing high-level features. For instance in a recognizing image system first edges and textures are recognized to later form shapes and objects. This hierarchical learning feature makes DNNs extremely powerful for tasks such as image and speech recognition, natural language processing, and even playing strategic games. Training DNNs, however, requires large amounts of data and computational power, and often employs many different techniques such as dropout and batch normalization to improve performance and prevent overfitting.

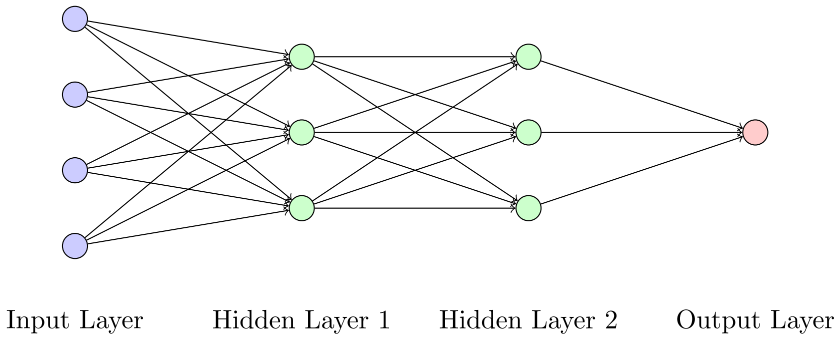


Figure 22: A Deep neural network with two hidden layers.

3.3.2 Convolutional Neural Networks

A *convolutional neural network* (CNN) [Figure 23] is a specialized type of deep neural network designed to process structured grid data, like images. At the core of CNNs there is the convolution operation which consists in sliding a set of filters over the input grid spatial data and consists in integrating two functions to produce a third one which expresses how the shape of one is modified by the other. Convolutions are performed in each position the filter slides on and typically involves a dot product followed by a summation in order to extract features. Mathematically, the convolution operation for a single filter K applied to an input I can be expressed as:

$$S(i,j) = (I * K)(i,j) = \sum_m \sum_n I(i-m, j-n)K(m,n)$$

Convolutional layers are typically followed by pooling layers, which are used to reduce the spatial dimensions of the data by typically halving it at each pass. Even tho it might seems deleterious it has been shown that applying pooling does not reduce performance while decrease computational complexity. CNNs have revolutionized computer vision tasks, achieving state-of-the-art results in image classification, object detection, and segmentation.

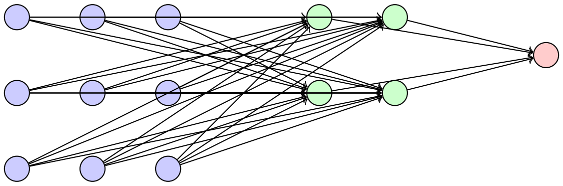


Figure 23: A simple convolutional neural network architecture.

3.3.3 Recurrent Neural Networks

A *recurrent neural network* (RNN) [Figure 24] is a specialized type of deep neural network that is particularly well-suited for sequential data, such as time series or natural language. Usually, when dealing with data in which the order does matter RNNs are used because they have connections that form directed cycles between its neurons, allowing information to persist. The hidden state h_t at time step t is computed based on the input x_t and the previous hidden state h_{t-1}:

$$h_t = \sigma(W_h h_{t-1} + W_x x_t + b)$$

where W_h and W_x are weight matrices, b is a bias vector, and σ is an activation function. A common problem with RNNs is the vanishing gradient problem which occurs when the calculated gradients become too small as they are backpropagated through long sequences. Variants of RNNs, such as Long Short-Term Memory (LSTM) and Gated Recurrent Unit (GRU) networks, try to mitigate this kind of issue while still allowing for learn long-term dependencies to be learned.

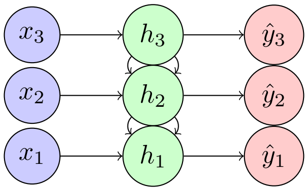


Figure 24: Recurrent Neural Network (RNN) unrolled through time.

3.3.4 Attention Mechanisms

In many contexts, it might be useful to focus more on specific input's parts than others and this is achieved through the usage of attention mechanisms which where firstly introduced in 2017 with the paper *Attention is all you need* [16]. In the context of text processing, each word in the input text is associated with a *key* and the element of focus is called *query*; then the attention mechanism [Figure 25] is assigning a *value* (weight) to each key with respect to the query. This allows the model to focus on important parts of the input in a dynamic manner and is especially useful in tasks involving sequences, such as machine translation and text summarization. The attention score for a query vector q and a set of key vectors $\{k_1, k_2, \ldots, k_n\}$ is computed as:

$$\text{Attention}(q, K, V) = \text{softmax}\left(\frac{qK^T}{\sqrt{d_k}}\right)V$$

where K is the matrix of keys, V is the matrix of values, and d_k is the dimension of the keys. Also worth to say, is that attention mechanism can be integrated in general with any type of neural network even though that's not always necessary.

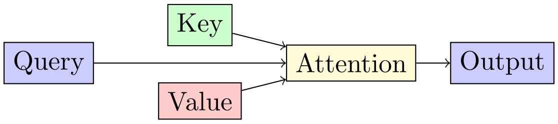


Figure 25: Attention mechanism in neural networks.

3.3.5 Graph Neural Networks

A *graph neural network* (GNN) is an advanced type of deep neural network designed to handle graph-structured data [section 2]. From social networks to molecules, from images to text manipulation, almost anything can be modelled as graphs. Hence, GNNs can be considered as one of the most powerful types of neural network architectures. The core concepts behind GNNs are the neighborhood aggregation and the message passing. The first is used to make a node aware of its neighborhood properties and the second to pass these informations through each node in the graph allowing GNNs to learn rich node representations which can be used for various tasks such as node classification, link prediction, and graph classification. The message-passing step for a node v can be mathematically expressed as:

$$h_v^{(k+1)} = \sigma\left(\sum_{u\in\mathcal{N}(v)} W h_u^{(k)} + b\right)$$

where $h_v^{(k+1)}$ is the node feature vector at layer $k+1$, $\mathcal{N}(v)$ denotes the neighbors of node v, W is a weight matrix, b is a bias vector, and σ is an activation function. There exists many variants of GNNs each leveraging different strategies on how to aggregate and update nodes informations such as Graph Convolutional Networks (GCNs), Graph Attention Networks (GATs), and Graph Recurrent Networks (GNRs).

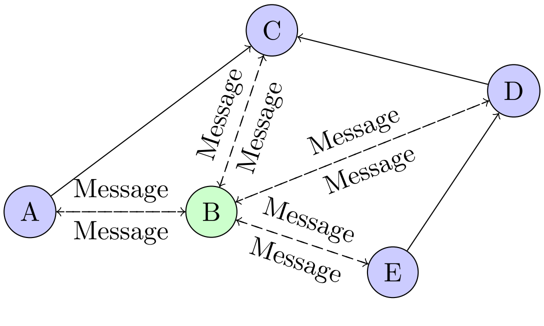


Figure 26: Message passing between node B and its neighbors in a GNN.

This work is specially focus on GNNs since every model in the state of the art is trying to address the graph edit distance problem [section 4], which make use of artificial neural networks to use this particular architecture with a Siamese layout.

4 Graph Similarity Problem

Of particular interest is to understand whether two given graphs are similar or not, this question takes the name of *graph similarity problem.* This problem could be of help in numerous domains and real world problems including pattern recognition, computer vision, bioinformatics, social network analysis, and chemical informatics. In such fields, common problems can be modelled as graphs and comparing the structure pair's properties of those could be very beneficial. For instance, in bioinformatics, comparing protein interaction networks can reveal functional similarities between different proteins, while in social network analysis, it can help identify similar community structures within different social groups. Since graph similarity is very important, numerous metrics have been developed to measure graph similarity, each with its own strengths and limitations to take into account. In the following sections, it will be explored several metrics commonly used to measure graph similarity: Graph Isomorphism, Graph Kernels, and Graph Edit Distance (GED). Each method will be discussed in terms of its fundamental concepts, applications, and limitations, with a focus on the latter. Also, it is often desirable to retrieve the edit path from one graph to another in a straightforward manner to understand the specific transformations involved. However, we will focus only on the similarity metrics and will not address the retrieval of edit paths.

4.1 Graph Isomorphism

In graph theory, graph isomorphism is one of the fundamental concepts used to determine if two graphs are structurally identical. Two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ are isomorphic if there is a bijection $f : V_1 \to V_2$ such that any two vertices u and v in G_1 are adjacent if and only if $f(u)$ and $f(v)$ are adjacent in G_2. Formally, G_1 and G_2 are isomorphic if:

$$(u, v) \in E_1 \Leftrightarrow (f(u), f(v)) \in E_2$$

Graph isomorphism metric provides in the a binary metric whether two graphs are identical in structure or not. Hence, it is limited because it does not quantify the degree of similarity at all. It is useful in scenarios where a binary outcome is desired. However, it is less useful in all the other cases where graphs are similar but not identical, as it cannot measure partial similarity or small structural differences.

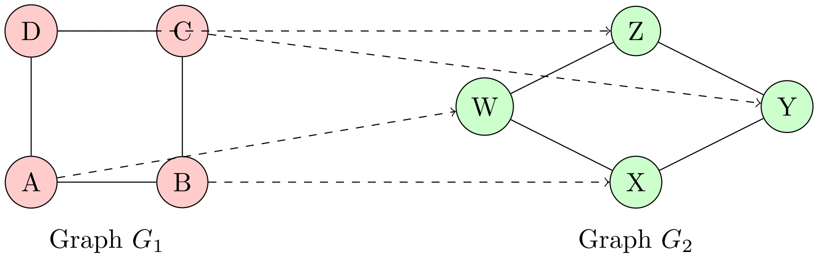


Figure 27: Graph Isomorphism: G_1 (Square) and G_2 (Rhombus).

4.2 Graph Kernels

A common solution in optimization theory when trying to separate two given dataset is to artificially increase their spatial dimension by using kernel tricks [11]. In the same way graph kernels transforms graphs into high-dimensional vectors where it is easier to compare them and exploit this mechanism to compute a similarity metric based on their structural attributes and properties. Common types of graph kernels include:

- **Random Walk Kernels**: Measure the similarity based on the number of matching random walks in both graphs.
- **Shortest Path Kernels**: Measure the similarity based on the distribution of shortest paths between pairs of nodes in each graph.
- **Weisfeiler-Lehman Kernels**: Measure the similarity utilizing an iterative node labeling algorithm to capture the neighborhood structure around each node.

Thus, structural information can be recovered in several different ways by utilizing graph kernels which can then be considered well-suited for use in machine learning algorithms where kernel tricks are commonly used to create algorithmic classificators. However, they can be computationally intensive if not carefully handled and also require careful tuning of parameters.

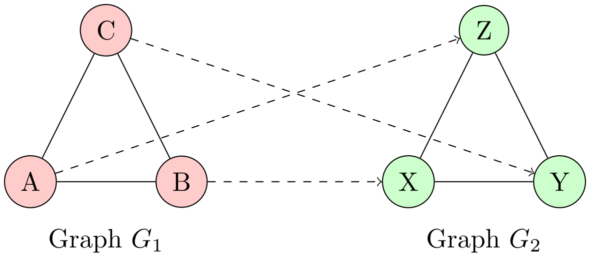


Figure 28: Graph Kernels: Example Graphs with Similar Structures.

4.3 Graph Edit Distance (GED)

One of the most flexible and informative metric that measure the similarity between two graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ is the *graph edit distance* (GED). It quantifies similarity by determining the minimum cost required to transform G_1 into G_2 by means of a series of atomic operations. These operations include but are not limited to vertex and edge insertions, deletions, and substitutions. The cost of each operation is determined by a predefined cost function which is usually 1.

Formally, let Σ be the set of all possible edit operations, and let $c : \Sigma \rightarrow \mathbb{R}^+$ be a cost function that assigns a positive real number to each operation. The GED, which falls in the range [0, inf), is then given by:

$$\text{GED}(G_1, G_2) = \min_{\sigma \in \Sigma^*} \sum_{o \in \sigma} c(o)$$

where Σ^* denotes the set of all finite sequences of operations from Σ, and o represents an individual operation within a sequence σ.

However, the computation of GED is known to be *NP-HARD* [1], which means that finding the exact minimum edit distance between two graphs is computationally intensive. Despite this, GED is preferred over other similarity metrics due to its flexibility and ability to provide a good measure of similarity even when the graphs are not identical.

The basic atomic operations in GED typically include:

- **Vertex Insertion**: Inserting a new vertex v into the graph.
- **Vertex Deletion**: Deleting an existing vertex v from the graph.
- **Vertex Substitution**: Replacing an existing vertex v with another vertex u.
- **Edge Insertion**: Inserting a new edge $e = \{u, v\}$ into the graph.
- **Edge Deletion**: Deleting an existing edge $e = \{u, v\}$ from the graph.
- **Edge Substitution**: Replacing an existing edge $e = \{u, v\}$ with another edge $e' = \{u', v'\}$.
- **Node Relabelling**: Replacing the label l of a vertex v with another label l'.

To illustrate the concept of Graph Edit Distance (GED), consider the pair of graphs represented in [Figure 29]:

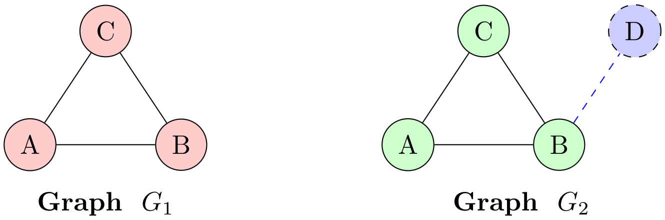


Figure 29: Graph Edit Distance: Transforming G_1 to G_2 by adding vertex D and edge (B,D).

In this example, graph G_1 has a vertex set $V_1 = \{A, B, C\}$ and an edge set $E_1 = \{\{A, B\}, \{A, C\}, \{B, C\}\}$, while graph G_2 has a vertex set $V_2 = \{A, B, C, D\}$ and an edge set $E_2 = \{\{A, B\}, \{A, C\}, \{B, C\}, \{B, D\}\}$. The transformation with the lowest cost from G_1 to G_2 involves inserting the vertex D and inserting the edge $\{B, D\}$. If we assign a cost of 1 to each operation, the total cost (GED) is: $1 + 1 = 2$.

5 State of the Art Review

Calculating the GED with traditional imperative algorithm is possible but feasible only for graphs of modest size. GED is a NP-HARD problems and for traditional solutions there is no way except to compare graphs node by node and edge by edge through combinatorial techniques to find a solution. However, as the number of nodes in the graphs increase, the complexity of these methods grows exponentially, leading to scalability issues thus infeasibility. To overcome these limitations recent works involve the use of artificial intelligence techniques such as neural networks to predict the GED between two graphs. AI-based approaches usually offer more robust and scalable solutions by learning patterns and features from graphs, significantly reducing computation times. This section reviews some of the most important papers dealing with GED calculation from 2019 to the time of writing this (2024).

A timeline of significant works, beginning in 2019 with *SimGNN* [3], the first to use neural networks for GED computation, is discussed in subsection 5.1. Following *SimGNN*, numerous models have been developed, each trying to offer something new and better performances. The timeline concludes with the most recent and promising model, *GedGNN* [12]. Although these models strive to estimate GED between graphs accurately, as will be shown, this area is still in the early stages of development.

5.1 Timeline

2019, *SimGNN: SimGNN: A Neural Network Approach to Fast Graph Similarity Computation* [3]: Presents the SimGNN to solve the problem of graph similarity using neural networks. It uses a learnable embedding function, an attention mechanism to capture important nodes, and a pairwise node comparison, which is more general and efficient than the baselines.

2020, *Learning Graph Edit Distance by Graph Neural Networks* [14]: Presents a framework that integrates deep metric learning with the conventional approximations of graph edit distance using geometric deep learning. The approach uses a message passing neural network (MPNN) to encode graph structure and compute graph distances effectively, achieving state-of-the-art performance in graph retrieval and competitive results in graph similarity learning.

2020, *Combinatorial Learning of Graph Edit Distance via Dynamic Embedding* [18]: Proposes a new method to solve the GED problem through a combination of dynamic graph embedding network and an edit path search method to improve the interpretability and the efficiency of the approach. The learning-based A* algorithm decreases the size of the search tree and time while providing a minor decrease in the solution quality.

2021, *Graph Partitioning and Graph Neural Network-Based Hierarchical Graph Matching for Graph Similarity Computation* [19]: Presents PSimGNN that first divides input graphs into subgraphs to learn local structural patterns and then employs a new GNN with attention to map subgraphs to embeddings and then combines coarse-grained interaction between subgraphs with fine-grained node-wise comparison to estimate similarity scores.

2021, *Noah: Noah: Neural Optimized A* Search Algorithm for Graph Edit Distance Computation* [20]: Presents Noah that uses A* search and GPN for approximate GED calculation. Noah estimates the cost function using GPN, includes pre-training with attention-based information, and uses an elastic beam size to decrease the search space.

2021, *Learning Efficient Hash Codes for Fast Graph-Based Data Similarity Retrieval* [17]: Proposes HGNN (Hash Graph Neural Network) that is a model for efficient graph-based data retrieval using GNNs and hash learning algorithms. HGNN learns a similarity-preserving graph representation and then generates short hash codes for efficient retrieval and classification.

2021, *More Interpretable Graph Similarity Computation via Maximum Common Subgraph Inference* [7]: Presents INFMCS, an end-to-end framework for graph similarity learning with an interpretable similarity score that is based on the correlation between the score and the Maximum Common Subgraph (MCS), and combines transformer encoder layers with graph convolution for high accuracy and interpretability.

2021, *H2MN: Graph Similarity Learning with Hierarchical Hypergraph Matching Networks* [21]: Presents H2MN, which computes the similarity of graph-structured data by converting graphs to hypergraphs and performing subgraph matching at the hyperedge level, and then a multi-perspective cross-graph matching layer.

2022, *TaGSim: Type-aware Graph Similarity Learning and Computation* [2]: This work introduces TaGSim, a type-aware graph similarity learning and computation approach that overcomes the drawbacks of traditional GED methods by incorporating type-specific graph edit operations. TaGSim models the effects of various graph modifications (node and edge insertions, deletions, and relabelings) as separate operations, which generate type-aware embeddings and use them for estimating the GED. The framework outperforms other GED solutions on real-world datasets as shown in the framework.

2023, *Efficient Graph Edit Distance Computation Using Isomorphic Vertices* [6]: Introduces a new strategy for the reduction of the search space of GED computation through the identification of isomorphic vertices, aiming at the elimination of unnecessary vertex mappings and thus a substantial reduction of the computation time for exact GED.

2023, *Exploring Attention Mechanism for Graph Similarity Learning* [15]: Introduces a single model with attention mechanisms for node embedding, cross-graph co-attention for interaction modeling, and graph similarity matrix learning for score prediction and outperforms the state of the art on benchmark datasets.

2023, *Graph Edit Distance Learning via Different Attention* [10]: Proposes DiffAtt, a new graph-level fusion module for GNNs to compute GED efficiently with the help of structural differences between graphs using attention, integrated into the GSC model REDRAFT, which outperforms the state of the art on benchmark datasets.

2023, *Graph-Graph Context Dependency Attention for Graph Edit Distance* [5]: Presents GED-CDA, a deep network architecture for GED computation which uses a graph-graph context dependency attention module that combines cross-attention and self-attention layers to model inter-graph and intra-graph dependencies.

2023, *GREED: A Neural Framework for Learning Graph Distance Functions*: Introduces GREED, a siamese GNN for learning GED and SED in a property preserving manner which outperforms other methods in terms of accuracy and time complexity.

2023, *MATA*: Learnable Node Matching with A* Algorithm for Approximate Graph Edit Distance* [9]: Presents MATA*, a novel approach for the approximate GED computation that combines GNNs and the A* algorithm, with the focus on learning the node matching.

2023, *Multilevel Graph Matching Networks for Deep Graph Similarity Learning* [8]: Introduces MGMN, a multilevel graph matching network that can capture the cross-level interactions, which includes NGMN and a siamese GNN for global-level interactions, and performs well when graph sizes are large.

2023, *Wasserstein Graph Distance Based on L1-Approximated Tree Edit Distance Between Weisfeiler-Lehman Subtrees* [4]: Introduces the WWLS distance which integrates WL subtrees with L1-TED which is more sensitive to fine changes in the structure of graphs and outperforms other methods in metric validation and graph classification.

2023, *Computing Graph Edit Distance via Neural Graph Matching* [12]: Presents GEDGNN, a deep learning model for GED computation that works on the idea of graph transformation instead of directly predicting GED value. GEDGNN provides GED values and a matching matrix and a post-processing procedure for obtaining high quality node mappings.

5.2 SimGNN

The first innovative model that used neural networks is SimGNN [3], introduced in 2019. SimGNN serves as a foundational model in the field of graph similarity computation, in fact future models will often inherit its core concepts (such as the siamese layout architecture), making it the starting point of reference for anyone dealing with GED computation.

The architecture of SimGNN [Figure 30] is composed by several stages:

- **Node Embedding Stage**: This stage makes use of a graph convolutional network to capture local structural information that transforms each node in the graph into a vector that encodes its features and structural properties.
- **Graph-Level Embedding Stage**: This stage produces a single embedding representing the whole graphs starting from the previously produced nodes embeddings by also using attention mechanisms to focus on important nodes.
- **Graph-Graph Interaction Stage**: This stage puts in communication the two graphs embedding previously produced and produces a matrix of similarity interaction scores.
- **Final Similarity Score Computation Stage**: This stage process the previously produced similarity matrix to compute the final similarity score.

In addition to the graph-level embedding interaction strategy, SimGNN has a its disposal a pairwise node comparison strategy:

- **Pairwise Node Comparison**: This strategy involves computing pairwise interaction scores between the node embeddings of the two graphs. The resulting similarity matrix is used to extract histogram features, which are then combined with the graph-level interaction scores to provide a comprehensive view of graph similarity.

The combination of these two strategies should allow the model to capture both global and local informations which should result in a robust approach to graph similarity computation.

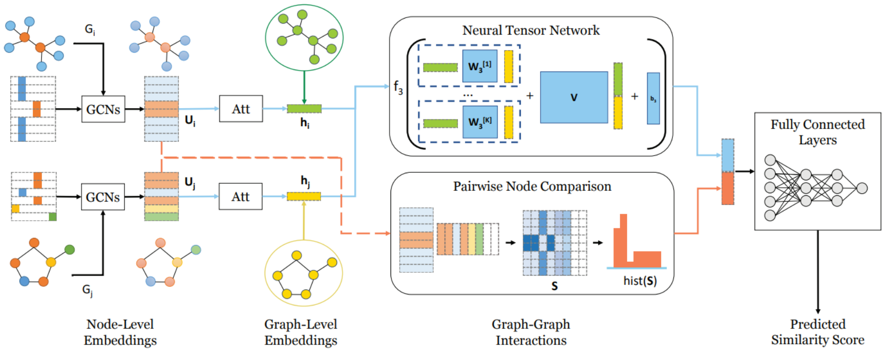


Figure 30: SimGNN architecture overview taken from [3].

5.3 GPN

In 2022, an innovative hybrid approach for computing GED was released. The Graph Path Networks (GPN) model, proposed within the *NOAH Framework* [20], introduces the GED computation by exploiting the A* search algorithm optimized through neural networks. This method tries to address several previously found limitations trying to improve both the search direction and search space optimization.

The architecture of GPN [Figure 31] is composed by several modules:

- **Pre-training Module**: This module computes pre-training information about the graphs that will be exploited by the next modules.
- **Graph Embedding Module**: This module utilizes layers of Graph Isomorphism Network (GIN) to transform each node into a vector. Then these embeddings are combined into a single graph level embedding by using different attention mechanisms.
- **Learning Module**: This module focuses on optimizing the A* search algorithm by learning an estimated cost function and an elastic beam size. The tradition algorithm is then used for the final prediction.

The main advantage of GPN over SimGNN is that it is capable of finding an edit path between graphs (roughly accurate) between graphs in a short amount of time.

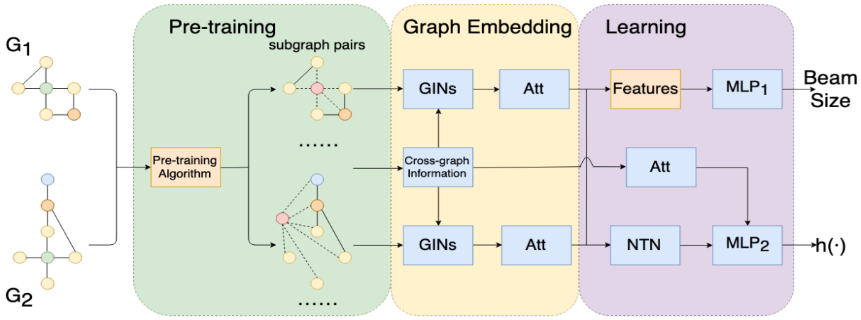


Figure 31: GPN architecture overview taken from [20].

5.4 TaGSim

In 2022, another innovative approach was released with TaGSim (Type-aware Graph Similarity) [2]. The idea behind GED as a single value has been revaluated and it is now thought as the summation of three different values: *ged_nc* the number of node relabelling, *ged_in* the number of node insertions/deletions, *ged_ie* the number of edges insertions/deletions.

The architecture of TaGSim [Figure 32] is composed by several components:

- **Type-Aware Graph Embeddings**: This component takes into account the different impacts that different atomic operations could have when predicting the GED producing a type-aware graph level embedding. Namely the operations taken into accounts are: node insertion/deletion (NR), node relabeling (NID), edge insertion/deletion (ER), and edge relabeling (EID). Each type of operation is handled separately to capture its localized effects on the graph.

- **Type-Aware Neural Networks**: This component takes advantage of specific neural networks that are specifically designed to process and learn from the type-aware embeddings. This allows TaGSim to achieve high accuracy in GED estimation by incorporating the distinct impacts of different edit types and outputs them all.

The main advantage of TaGSim over predecessors is that by decoupling the GED into different dimensions, there is the potential for more granular control and learnability.

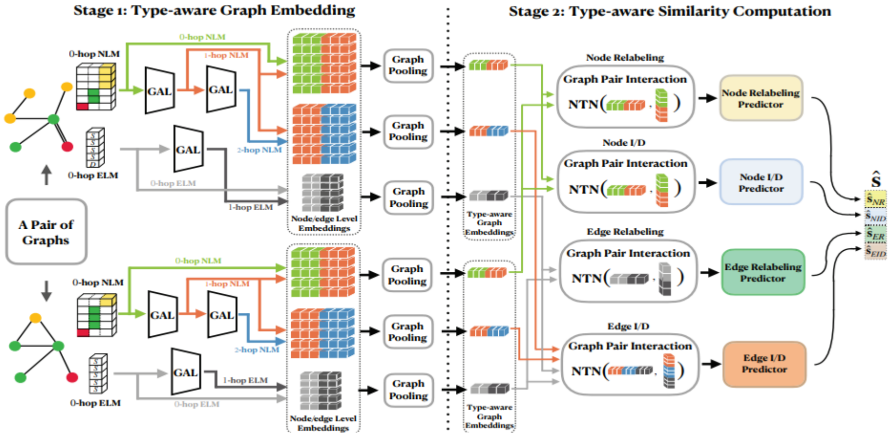


Figure 32: TaGSim architecture overview taken from [2].

5.5 GedGNN

In 2023, the model that is considered the state of the art at the time of writing this (2024) is released with GedGNN (Graph Edit Distance via Neural Graph Matching) [12]. The idea behind this model is to try to put together all the best ideas from past's models including the basic siamese layout of SimGNN, the use of more advanced convolutional layers of GPN and the split of the GED metric from TaGSim while still allowing for the retrieval of an edit paths by taking inspiration from NOAH framework.

The architecture of GedGNN [Figure 33] is composed by several components:

- **Graph Neural Network (GNN) Encoder**: This component produces the encodings for nodes and edges while preserving their relational information. This is done through the employment of an advanced GNN encoder.
- **Node and Edge Matching Module**: This component performs the node and edge matching between the pair of graphs producing a matching matrix and a cost matrix.
- **k-Best Matching Post-Processing Algorithm**: After predicting the GED value a k-best post-processing algorithm is used trying to retrieve a good edit path.

GedGNN's results state to not only outperforms previous methods but also provides a flexible framework that can adapt to various types of graph structures and similarity measures.

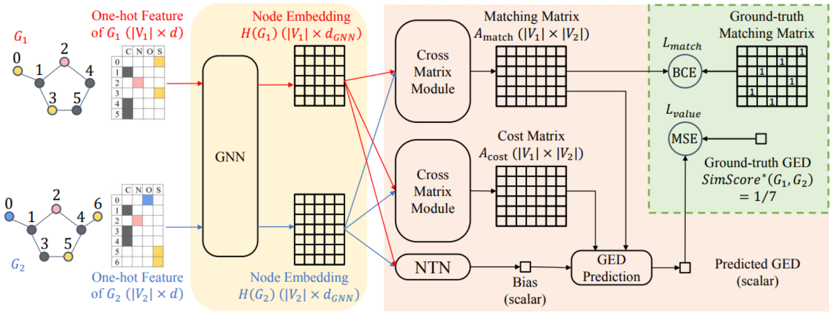

G_1
One-hot Feature of G_1 ($|V_1| \times d$)
C N O S
Node Embedding $H(G_1)$ ($|V_1| \times d_{GNN}$)
GNN
G_2
One-hot Feature of G_2 ($|V_2| \times d$)
Node Embedding $H(G_2)$ ($|V_2| \times d_{GNN}$)
Cross Matrix Module
Matching Matrix A_{match} ($|V_1| \times |V_2|$)
Cross Matrix Module
Cost Matrix A_{cost} ($|V_1| \times |V_2|$)
NTN
Bias (scalar)
GED Prediction
Predicted GED (scalar)
L_{match}
BCE
Ground-truth Matching Matrix
L_{value}
MSE
Ground-truth GED $SimScore^*(G_1, G_2) = 1/7$

Figure 33: GedGNN architecture overview taken from [12].

5.6 Encountered Gaps

In this research, some problems and concerns have been identified from the codebase of GedGNN. Most of these issues can be attributed to the fact that codes are copied from one project to another and reused by researchers without proper testing.

The first problem is related to the code quality of the codebase. It is often poor and there is also no strong adherence to best practices. There are numerous method implementations that look very much like they do the same thing, too many classes, numerous questions, and no way of handling the errors. In some parts, the code appears to have been almost hard coded, thus, offers little room for modification. Also, the code can be hardly understood, and the comments that are given with the code are quite uninformative.

Closely related to code quality and testing scenarios is the problem of scalability on GPUs. When trying to run for the first time the code as is on a GPU some errors also raised up, but apart from this, the models do not scale well on GPU hardware and this is an issue. Training on small datasets is possible on CPUs but when it comes to using large datasets for testing and training new models, issues arise with long training time per epoch.

Another concern is that the codebase only permits the use of graphs with 10 nodes or less in training, testing, and validation cases. This could be attributed to the fact that there are complex imperative algorithms that are used in the software such as the k-best post-processing algorithm for reconstructing edit paths. In addition, for any model, the code fails when tested on the IMDB dataset. Additionally, despite the presence of artificial dataset generation code, it is not utilized, leading to confusion and unreachable code paths.

The training data, however, has issues that have been there from the time SimGNN was created. The same datasets are used over and over again, however, the most important problem is that approximate GEDs are used as labels instead of the exact one. These datasets are relatively small; graphs contain less than ten nodes, and with today's hardware, we would not spend much time calculating exact GED. Also, in all codebases analyzed so far, the training is done in a way that for each pair of graphs in the training set there is a corresponding GED.

However, the biggest problem has to do with the fact that there is no fair testing between models. It is still not well understood why so, but with the current state of the codebase, one cannot even attempt to test a model trained on one dataset with another dataset. This restriction may be to hide the fact that there is no generalization capability.

In summary, the analysed codebase which presents GedGNN, TaGSim, SimGNN and GPN presents several significant challenges to overcome. These include problems with the reproducibility of the results, fair evaluations, scalability issues, poor code quality, unclear parameters and more. Clearly, resolving many of this issues would lead to a significant advancement in this field.

6 Methodology and Experimentation

Working with high-quality datasets is crucial for developing performant models. However, in [12], the data, particularly the GED labels, are mostly approximations, which negatively impacts model training and performance. A method for generating artificial data is detailed in [subsection 6.1], while the results of fair testing are summarized in the tables found in [subsection 6.2]

6.1 Artificial Dataset Generation

The proposed code is publicly available on GitHub and is meant as a starting point for generating an high quality dataset composed of exact (TaGSim like) GED values along with randomly generated graphs at fixed distances. When working with Python programming language it is often not needed to reinvent the wheel because a package that suits requirements probably already exists and within this context, a large use of NetworkX (nx) is made to handle graphs data structures. Specifically, two methods for generating an artificial dataset are proposed: one does it in an incremental way, starting from a small graph and increasing its complexity with each iteration, the other does it in the opposite way, starting from a big graph and reducing its complexity at each step. With such methodology it will be possible to generate a dataset of n graphs and know the GED for each pair of graph in the dataset.

Listing 1: Random Graph Generator

```
class RandomGraphGenerator():
  def generate_random_ER_graph(self, nmin=2, nmax=10, pmin=0.2, pmax=1):
    """Erdos-Renyi graph generation"""
    n  = randint(nmin, nmax)
    p = uniform(pmin, pmax)
    G = nx.gnp_random_graph(n, p, seed=None, directed=False)
    return self._make_connected(G)

  def generate_random_BA_graph(self, nmin=2, nmax=10):
    """Barabasi Albert graph generation"""
```

```
        n  = randint(nmin, nmax)
        m = randint(1, n-1)
        if not (m >= 1 and m < n):
            raise Exception(f"m >= 1 and m < n", f"m={m}", f"n={n}")
        G = nx.barabasi_albert_graph(n, m)
        return self._make_connected(G)

    def _make_connected(self, G : nx.Graph):
        for node in list(nx.isolates(G)):
        target_node = choice([n for n in G.nodes() if n != node])
        G.add_edge(node, target_node)
        return G
```

RandomGraphGenerator [Listing 1] is a class that provides two public methods for generating random graphs by reusing *nx* package. Specifically two methods for generating different variants of graphs are provided: one for Erdős-Rényi's types of graphs (or binomial graph) and one for Barabási–Albert's ones. Additionally, it is important to make sure that graphs are connected at any time to prevent errors during future computations.

Listing 2: Abstract Consecutor

```
class Consecutor(ABC):
    def next(self, G : nx.Graph) -> HistoryValue:
        """Return a tuple with a new graph G’ and the distance from G (can
            be 0)"""
        if not self._is_processable(G):
            raise UnprocessableError()
        copy = deepcopy(G)
        rand = random()
        return self._next(copy, rand)

    @abstractmethod
    def _next(self, G : nx.Graph, rand : float) -> HistoryValue:
        """Actual next logic from concrete classes"""
        raise NotImplementedError()

    def _is_processable(self, G : nx.Graph) -> bool:
        """Whether you can make a ’next’ on graph G"""
        return len(self._nodes(G)) > 0 and len(self._edges(G)) > 0

    def _nodes(self, G : nx.Graph) -> List[int]:
        """Return the list of nodes of the graph G"""
        return list(G.nodes)

    def _edges(self, G : nx.Graph) -> List[Tuple[int, int]]:
        """Return the list of edges of the graph G"""
        return list(G.edges)

    def _rand_obj_list(self, l : List):
        """Return a random object in list if not empty else None"""
        return l[randint(0, len(l) - 1)] if len(l) > 0 else None

    def _new_node(self, G : nx.Graph):
        """Return a new node for the graph G (biggest indexed node + 1)"""
        return (self._nodes(G)[-1] + 1) if len(self._nodes(G)) > 0 else 0

    def _rand_node(self, G : nx.Graph):
        """Return a random existing node of G if any else None"""
        return self._rand_obj_list(self._nodes(G))

    def _new_edge(self, G : nx.Graph):
        """Return a new edgre for the graph G if not fully-connected else
            None"""
        return self._rand_obj_list(list(nx.non_edges(G)))

    def _rand_edge(self, G : nx.Graph):
        """Return a random existing edge of G if any else None"""
        return self._rand_obj_list(self._edges(G))
```

The *Consecutor* class is what will handle the generation of a graph G' starting from G at a fixed known GED distance. In [Listing 2] is showed the abstract *Consecutor* class that provides common methods for managing graphs modifications.

Listing 3: Incremental Consecutor

```
class IncrementalConsecutor(Consecutor):
   """IncrementalConsecutor add nodes and edges. The way it does so
       ensures there are no isolates at any moment."""
   def _next(self, G : nx.Graph, rand : float) -> HistoryValue:
      if rand <= 0.7:
         return self.__add_edge(G)
      else:
         return self.__add_node_and_edges(G)

   def __add_node_and_edges(self, G : nx.Graph) -> HistoryValue:
      """Add a new node and k edges from the new node to random nodes"""
      new_node = super()._new_node(G)
      G.add_node(new_node)
      nodes = super()._nodes(G)
      k = randint(1, len(nodes) - 1)
      choices = list(filter(lambda n : n != new_node, nodes))
      for _ in range(0, k):
         target = super()._rand_obj_list(choices)
         G.add_edge(new_node, target)
         choices.remove(target)
      return G, (1+k, 0, 1, k)

   def __add_edge(self, G : nx.Graph) -> HistoryValue:
      """Add a new edge if not fully connected"""
      new_edge = super()._new_edge(G)
      if new_edge is None:
         return G, (0, 0, 0, 0)
      G.add_edge(*new_edge)
      return G, (1, 0, 0, 1)
```

The *IncrementalConsecutor* [Listing 3] is the class responsible for creating a graph G' from G in an additive way. If a graph G_1 is at distance d_1 from G_2 and G_3 is generated by solely adding edges or nodes with distance d_2 from G_2 then it is demonstrable that G_1 is distant $|d_1 + d_2|$ from G_3.

Listing 4: Decremental Consecutor

```
class DecrementalConsecutor(Consecutor):
"""DecrementalConsecutor removes nodes and edges, after any atomic
    operation it also removes isolated nodes."""
    def _next(self, G : nx.Graph, rand : float) -> HistoryValue:
        if rand <= 1:
            return self._remove_edge(G)
        else:
            return self._remove_node_and_edges(G)

    def _remove_node_and_edges(self, G : nx.Graph) -> HistoryValue:
        """Remove a random node along with its edges, if this causes a
            node to be isolated it is removed aswell"""
        rvm_node = self._rand_node(G)
        if rvm_node is None:
            return G, (0, 0, 0, 0)
        degree = G.degree(rvm_node)
        G.remove_node(rvm_node)
        isolated = list(nx.isolates(G))
        G.remove_nodes_from(isolated)
        return G, (1+degree+len(isolated), 0, 1+len(isolated), degree)

    def _remove_edge(self, G : nx.Graph) -> HistoryValue:
        """Remove a random edge if there are any, if this causes a node to
            be isolated it is removed aswell"""
        rvm_edge = self._rand_edge(G)
        if rvm_edge is None:
            return G, (0, 0, 0, 0)
        G.remove_edge(*rvm_edge)
        isolated = list(nx.isolates(G))
        G.remove_nodes_from(isolated)
        return G, (1+len(isolated), 0, len(isolated), 1)
```

At the contrary, *DecrementalConsecutor* [Listing 4] is the class responsible for creating a graph G' from G in an subtractive way. If a graph G_1 is at distance d_1 from G_2 and G_3 is generated by solely removing edges or nodes with distance

d_2 from G_2 then it is demonstrable that G_1 is distant $|d_1 + d_2|$ from G_3.

Listing 5: Consecutor Executor

```
class ConsecutorExecutor():
    """ConsecutorExecutor can be used to execute steps consecutions
        starting from a graph G"""
    def __init__(self, consecutor: Consecutor):
        self.consecutor = consecutor

    def execute(self,
        G : nx.Graph,
        steps = 100,
        stopper : Callable[[nx.Graph], bool] = None,
        skip_zero_ged = True) -> History :
        """Perform steps attempts to modify graph G.
        Parameters:
        1. G, the graph where to start from
        2. steps, the number of atomic modifications
        3. stopper, an early custom stopping function on newly generated
            graph
        4. skip_zero_ged, a Consecutor may return a G' with ged 0 w.r.t. G
        Returns a dict representing the history of graph generations with
            edit distance from previous graph.
        """
        history = {}
        history[0] = (G, (0, 0, 0, 0))
        for i in tqdm(range(1, steps+1), total=steps+1, desc="History
            Generation"):
            try:
                # Generation and update G
                G, taged = self.consecutor.next(G)
            except UnprocessableError:
                break
        # Custom stopping condition on newly generated graph
        if stopper is not None and stopper(G):
            break
        # Save only when necessary
        if taged[0] != 0 or not skip_zero_ged:
            history[i] = (G, taged)
        return history
```

ConsecutorExecutor [Listing 5] is the class that handles the *generatio* of *steps* sequential graphs by applying the *next* method from either the *IncrementalConsecutor* or *DecrementalConsecutor*. It is not possible to apply both in the same sequence as the GED value will be invalidated between pairs of graphs: a drawback of this approach is the unfeasibility of building dense and very large datasets.

Listing 6: History Utilities

```
class HistoryUtilities():
   """HistoryUtilities is responsible for providing common history
       utilities functions."""
   def build_ged_combination(self, history : History) ->
       List[MappingGed]:
      """Function that builds the ged pickle file from the combination
          of every pair of graphs in history"""
      mapping_list = []
      entries = list(history.items())
      all_combs = list(combinations(entries, 2))
      for comb in tqdm(all_combs, total=len(all_combs), desc="Ged Dict
          Generation"):
         id1, id2 = comb[0][0], comb[1][0]
         value = self.calculate_ged_comb(history, id1, id2)
         mapping_list.append(value)
      return mapping_list

   def calculate_ged_comb(self, history : History, id1 : HistoryKey,
       id2: HistoryKey) -> MappingGed:
      """Returns the artificial ged distance given an entry
          combination"""
      delimiters = [id1, id2]
      delimiters.sort()
      min, max = delimiters[0], delimiters[1]
      ged = ged_nc = ged_in = ged_ie = 0
      for entry in history.items():
         key = entry[0]
         value = entry[1]
         if min < key <= max:
            TaGED = value[1]
            ged += TaGED[0]
            ged_nc += TaGED[1]
            ged_in += TaGED[2]
            ged_ie += TaGED[3]
         if key > max:
            break
      return (id1, id2, ged, ged_nc, ged_in, ged_ie, [])

   def split_by_fractions(self, history: History, train=0.8, test=0.2):
      """Split history in two dicts according to proportions"""
      assert train+test==1.0, 'fractions sum is not 1.0'
      keys = list(history.keys())
      shuffle(keys)
      split_point = int(train * len(keys))
      dict_train = {key: history[key] for key in keys[:split_point]}
      dict_test = {key: history[key] for key in keys[split_point:]}
      return dict_train, dict_test
```

```
  def save_to_sparse_jsons(self, history : History, outfolder : str):
    """Create/Clean outfolder than save all history as jsons files"""
    if not os.path.exists(outfolder):
      os.makedirs(outfolder)
    file_list = os.listdir(outfolder)
    jsons_files = [file for file in file_list if
        file.endswith('.json')]
    for file in jsons_files:
      file_path = os.path.join(outfolder, file)
      os.remove(file_path)
    for key, value in tqdm(history.items(),
        total=len(history.items()), desc=f"Saving to {outfolder}"):
      filename = os.path.join(outfolder, f'{key}.json')
      with open(filename, 'w') as f:
        graph = {}
        graph['n'] = value[0].number_of_nodes()
        graph['m'] = value[0].number_of_edges()
        graph['labels'] = None
        graph['graph'] = list(list(map(lambda t: list(t),
            value[0].edges)))
        json.dump(graph, f)
```

HistoryUtilities [Listing 6] is the final piece of utility to generate and save to disk the dataset consisting of all possible combinations of a given list of sequentially generated graphs along with their TaGED.

Listing 7: Dataset Generation Example

```
generator = RandomGraphGenerator()
hist_utils = HistoryUtilities()
stop_on_empty = lambda G: len(list(G.nodes))==0

start = generator.generate_random_ER_graph(3, 6, 0.5, 1)
# start = generator.generate_random_BA_graph(2, 5)

consecutor = IncrementalConsecutor()
# consecutor = DecrementalConsecutor()

exc_consecutor = ConsecutorExecutor(consecutor)

# Change the parameter to generate more consecutio steps
history = exc_consecutor.execute(start, steps=1000,
    stopper=stop_on_empty, skip_zero_ged=True)

ged = hist_utils.build_ged_combination(history)
train, test = hist_utils.split_by_fractions(history, train=0.8, test=0.2)

NAME = "Medium"
hist_utils.save_to_sparse_jsons(train, f'json_data/{NAME}/train/')
hist_utils.save_to_sparse_jsons(test, f'json_data/{NAME}/test/')
json.dump(ged, open(f'json_data/{NAME}/TaGED.json', 'w'))
```

With [Listing 7] is an example of how to put all the pieces together to actually create a custom dataset and save it to disk in a format suitable GedGNN. With this specific code, it will be generated a dataset called *Medium* (more information in subsection 6.2) consisting of 1000 graphs, starting from random graph called *start* in an *incremental* way; then the data is processed to retrieve GED information for each pair of graphs and the whole is split into a training set (80%) and a test set (20%).

6.2 Experiments and Results

For experimenting and conducting a fair evaluation of the models, two well known datasets, IMDB and Linux, as well as two artificially generated datasets, *1000g_100n* and *Medium* have been employed:

- **Linux**: A dataset that consists of graphs representing function calls within the Linux kernel: a node represent a statement and edges represent the dependency between two statements. The dataset is composed of 1000 graphs and each of them does not have more than 10 nodes, making data specialized and dense.
- **IMDB**: A dataset that consists of movie-related graphs: a node represents an actor, while edges connects two actors if they appear in the same movie. The dataset is composed of 1500 graphs and some of them have more than 10 nodes, making data less dense with respect to *Linux*.
- **1000g_100n**: An artificially generated dataset with 1000 graphs, each containing 100 nodes and a progressively less number of edges. *1000g_100n* does not represent any real scenario in particular and has been generated solely for testing purposes.
- **Medium**: Another artificially generated dataset with 1000 heterogeneous graphs. The variety is big, starting from 3 nodes and 3 edges in the firstly generated graph up to 297 nodes and 21457 edges in the last one, making data very sparse.

Each model presented in the VLDB paper [12], namely SimGNN [subsection 5.2], GPN [subsection 5.3], TaGSim [subsection 5.4] and GedGNN [subsection 5.5] has been trained for 10 epochs with default parameters from the codebase as is for each of the aforementioned dataset. Then each of trained model has been tested, trying to reproduce [12] results on the respective test-set and **on all the other datasets as well** for a total of 64 combinations. In the next tables, results are presented with key metrics:

- **Mean Squared Error (mse)**: Measures the average of the squares of the differences between the predicted GED values and real GED values.

- **Mean Absolute Error (mae)**: Measures the average differences between the predicted GED values and the reak GED values.
- **Accuracy (acc)**: Measures the proportion of correct GED predictions over the total predicted GEDs.

Since predicting GED 100% accurately is very hard for a neural network, the most relevant metric in this context will be the MAE: it is important that the net is very close to the real value.

trainset	**testset**	**mse**	**mae**	**acc**
IMDB	IMDB	4532	**1.28**	0.475
IMDB	Linux	105097	6506	0.008
IMDB	1000g_100n	725792	327867	0.005
IMDB	Medium	362746	5527745	0.004
Linux	IMDB	119051	7414	0.202
Linux	Linux	2547	**0.423**	0.64
Linux	1000g_100n	725792	327867	0.005
Linux	Medium	390191	5996899	0.004
1000g_100n	IMDB	123898	5104	0.043
1000g_100n	Linux	135375	5938	0.027
1000g_100n	1000g_100n	66845	219972	0.002
1000g_100n	Medium	40.19	6938538	0.0
Medium	IMDB	81696	**5.25**	0.057
Medium	Linux	62777	5075	0.079
Medium	1000g_100n	531549	314.44	0.002
Medium	Medium	2436	2724294	0.001

Table 1: Results for SimGNN models

As show in [Table 1], SimGNN works well if it gets trained on a specific dataset and tested on the same type of data; unfortunately there are no significant results for artificial generated datasets, but clearly the model does not generalize. *SimGNN* trained on *Medium* seems to show positive outcomes on *IMDB* but it might due to the fact that few identical already seen graphs could have been feed to the net.

trainset	**testset**	**mse**	**mae**	**acc**
IMDB	IMDB	106.09	**10645**	0.211
IMDB	Linux	41.77	2762	0.089
IMDB	1000g_100n	6936	327867	0.005
IMDB	Medium	502857	7301305	0.005
Linux	IMDB	44103	5777	0.117
Linux	Linux	57506	**3306**	0.065
Linux	1000g_100n	101895	1476.37	0.0
Linux	Medium	155102	3926197	0.0
1000g_100n	IMDB	80138	8048	0.118
1000g_100n	Linux	62778	3251	0.052
1000g_100n	1000g_100n	180992	1979288	0.0
1000g_100n	Medium	110796	3312005	0.0
Medium	IMDB	85028	8398	0.061
Medium	Linux	60467	3275	0.052
Medium	1000g_100n	340701	2184379	0.0
Medium	Medium	276667	5131201	0.001

Table 2: Results for GPN models

GPN seems to be an outlier, since reproducing its original performance has not been possible. As shown in [Table 2], the model does perform bad in every associated scenario.

trainset	**testset**	**mse**	**mae**	**acc**
IMDB	IMDB	**5.2**	2743	0.183
IMDB	Linux	55426	6.42	0.006
IMDB	1000g_100n	34168	250498	0.002
IMDB	Medium	109842	6867463	0.0
Linux	IMDB	104002	185839	0.0
Linux	Linux	1408	**0.427**	0.642
Linux	1000g_100n	33935	671133	0.0
Linux	Medium	89117	6445506	0.0
1000g_100n	IMDB	179554	12471	0.001
1000g_100n	Linux	193506	16509	0.0
1000g_100n	1000g_100n	22926	225854	0.002
1000g_100n	Medium	94807	7074655	0.0
Medium	IMDB	101271	7366	0.155
Medium	Linux	114007	11265	0.0
Medium	1000g_100n	64129	671133	0.0
Medium	Medium	5757	6446658	0.0

Table 3: Results for TaGSim models

TaGSim seems to show the same behaviour as *SimGNN* performing good only when tested on the same type of graphs on which it got trained. A Lack of generalization capability is shown here as well.

trainset	**testset**	**mse**	**mae**	**acc**
IMDB	IMDB	0.816	**0.634**	0.58
IMDB	Linux	9399	1.27	0.221
IMDB	1000g_100n	13861	363687	0.005
IMDB	Medium	201372	4106777	0.005
Linux	IMDB	13153	3539	0.075
Linux	Linux	1161	**0.315**	0.735
Linux	1000g_100n	869809	4367593	0.0
Linux	Medium	212754	3801531	0.0
1000g_100n	IMDB	47711	**5.86**	0.033
1000g_100n	Linux	28688	2126	0.105
1000g_100n	1000g_100n	0.708	78892	0.013
1000g_100n	Medium	411299	6885001	0.003
Medium	IMDB	65417	7611	0.103
Medium	Linux	9204	**1.17**	0.261
Medium	1000g_100n	4651	259151	0.003
Medium	Medium	0.718	267837	0.003

Table 4: Results for GedGNN models

GedGNN seems to be the most promising model for artificial dataset testing. When training on *1000g_100n* and *Medium*, *GedGNN* showed positive results while getting tested on both *IMDB* and *Linux*. A clear sign that if a much bigger and denser dataset were to be used it could have outperformed others specialized types of datasets.

In conclusion, it is necessary to differentiate between the testing of models on the same data distribution that has been used during training (in-distribution) and testing on new types of data (out-of-distribution). When the models are trained and tested on similar types of graph data (in-distribution), most of the models do well with GedGNN being the best. However, when models are evaluated on data that is not part of the training distribution, the performance of all the models is rather poor. There is one exception, which is the models trained on the Medium dataset, which has the potential of enhancing the generalization but the dataset needs to be denser and much larger. At the moment, heuristic methods are usually more effective than models in out-of-distribution conditions, especially when evaluating deviation from the true GED (Graph Edit Distance).

7 Discussion and Conclusions

In this thesis, the Graph Edit Distance (GED) has been introduced and explained—a concept which is important in graph theory and which defines the cost of transformation of one graph into another. This is important considering that graphs are used in various fields including bioinformatics, social network analysis among others where proper and fast GED determination is crucial. Although new developments have been made in deep learning especially Graph Neural Networks (GNNs), most of the current methods do not have good generalization capabilities.

The content of this paper shows that the current neural network-based solutions are quite innovative but they are associated with small and approximate datasets. It has been established that the usage of artificially generated datasets is promising; nevertheless, the current findings suggest that the size and density of such datasets remain limited, which hinders the development of models with the potential for generalization. More research should be directed towards the creation of bigger and more dense synthetic graphs similar to real graphs which would improve the performance of the models.

In conclusion, it is possible to state that the current state of research on neural network-based GED computation has been significantly improved, yet, there is still a large discrepancy between the current state of research and the real-world needs for effective, efficient, and robust GED computation. In order to meet these challenges, it will be necessary to develop new and more complex models and, at the same time, to gather larger and more detailed datasets. This will be important for achieving the best results in the application of GED in numerous areas such as in drug development, analysis of social networks, and so on, and therefore enhance the effectiveness of solutions in these spheres.

References

[1] Eric Allender and Michael Loui. Complexity classes. 07 2003.

[2] Jiyang Bai and Peixiang Zhao. Tagsim: type-aware graph similarity learning and computation. *Proceedings of the VLDB Endowment*, 15:335–347, 02 2022.

[3] Yunsheng Bai, Hao Ding, Song Bian, Ting Chen, Yizhou Sun, and Wei Wang. Simgnn: A neural network approach to fast graph similarity computation. page 384–392, 2019.

[4] Zhongxi Fang, Jianming Huang, Xun Su, and Hiroyuki Kasai. Wasserstein graph distance based on l1–approximated tree edit distance between weisfeiler–lehman subtrees. *Proceedings of the AAAI Conference on Artificial Intelligence*, 37(6):7539–7549, Jun. 2023.

[5] Ruiqi Jia, Xianbing Feng, Xiaoqing Lyu, and Zhi Tang. Graph-graph context dependency attention for graph edit distance. pages 1–5, 2023.

[6] Jongik Kim. Efficient graph edit distance computation using isomorphic vertices. *Pattern Recognition Letters*, 168:71–78, 2023.

[7] Z. Lan, B. Hong, Y. Ma, and F. Ma. More interpretable graph similarity computation via maximum common subgraph inference. *IEEE Transactions on Knowledge; Data Engineering*, (01):1–12, apr 2021.

[8] Xiang Ling, Lingfei Wu, Saizhuo Wang, Tengfei Ma, Fangli Xu, Alex X. Liu, Chunming Wu, and Shouling Ji. Multilevel graph matching networks for deep graph similarity learning. *IEEE Transactions on Neural Networks and Learning Systems*, 34(2):799–813, February 2023.

[9] Junfeng Liu, Min Zhou, Shuai Ma, and Lujia Pan. Mata*: Combining learnable node matching with a* algorithm for approximate graph edit distance computation. page 1503–1512, 2023.

[10] Jiaxi Lv, Liang Zhang, Yi Huang, Jiancheng Huang, and Shifeng Chen. Graph edit distance learning via different attention. 2023.

[11] Bernhard Olkopf. The kernel trick for distances. 02 2001.

[12] Chengzhi Piao, Tingyang Xu, Xiangguo Sun, Yu Rong, Kangfei Zhao, and Hong Cheng. Computing graph edit distance via neural graph matching. *Proc. VLDB Endow.*, 16(8):1817–1829, apr 2023.

[13] Rishabh Ranjan, Siddharth Grover, Sourav Medya, Venkatesan Chakaravarthy, Yogish Sabharwal, and Sayan Ranu. Greed: A neural framework for learning graph distance functions. 2023.

[14] Pau Riba, Andreas Fischer, Josep Lladós, and Alicia Fornés. Learning graph edit distance by graph neural networks. 2020.

[15] Wenhui Tan, Xin Gao, Yiyang Li, Guangqi Wen, Peng Cao, Jinzhu Yang, Weiping Li, and Osmar R. Zaiane. Exploring attention mechanism for graph similarity learning. *Know.-Based Syst.*, 276(C), sep 2023.

[16] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. 2023.

[17] Jinbao Wang, Shuo Xu, Feng Zheng, Ke Lu, Jingkuan Song, and Ling Shao. Learning efficient hash codes for fast graph-based data similarity retrieval. *IEEE Transactions on Image Processing*, 30:6321–6334, 2021.

[18] Runzhong Wang, Tianqi Zhang, Tianshu Yu, Junchi Yan, and Xiaokang Yang. Combinatorial Learning of Graph Edit Distance via Dynamic Embedding. *arXiv e-prints*, page arXiv:2011.15039, November 2020.

[19] Haoyan Xu, Ziheng Duan, Yueyang Wang, Jie Feng, Runjian Chen, Qianru Zhang, and Zhongbin Xu. Graph partitioning and graph neural network based hierarchical graph matching for graph similarity computation. *Neurocomputing*, 439:348–362, 2021.

[20] Lei Yang and Lei Zou. Noah: Neural-optimized a* search algorithm for graph edit distance computation. pages 576–587, 2021.

[21] Zhen Zhang, Jiajun Bu, Martin Ester, Zhao Li, Chengwei Yao, Zhi Yu, and Can Wang. H2mn: Graph similarity learning with hierarchical hypergraph matching networks. page 2274–2284, 2021.