

**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:  
Prof. Giorgio Terracina  
Prof. Sebastiano Antonio Piccolo

Candidate:  
Federico Calabrò  
Matricola 247976

---

Academic Year 2023/2024

# Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
<b>2</b>	<b>Graph Data Structure</b>	<b>5</b>
2.1	Types of Graphs . . . . .	5
2.2	Graph Representation . . . . .	9
2.3	Properties of Graphs . . . . .	10
<b>3</b>	<b>Neural Networks</b>	<b>12</b>
3.1	Basic Structure of a Neural Network . . . . .	12
3.1.1	Activation Functions . . . . .	13
3.2	Training Neural Networks . . . . .	14
3.2.1	Backpropagation . . . . .	14
3.2.2	Gradient Descent . . . . .	16
3.3	Advanced Topics in Neural Networks . . . . .	16
3.3.1	Deep Neural Networks . . . . .	16
3.3.2	Convolutional Neural Networks . . . . .	17
3.3.3	Recurrent Neural Networks . . . . .	18
3.3.4	Attention Mechanisms . . . . .	18
3.3.5	Graph Neural Networks . . . . .	19
<b>4</b>	<b>Graph Similarity Problem</b>	<b>20</b>
4.1	Graph Isomorphism . . . . .	20
4.2	Graph Kernels . . . . .	21
4.3	Graph Edit Distance (GED) . . . . .	22
4.3.1	Atomic Operations . . . . .	22
<b>5</b>	<b>State of the Art Review</b>	<b>23</b>
5.1	Timeline . . . . .	24
5.2	SimGNN . . . . .	26
5.3	GPN . . . . .	27
5.4	TaGSim . . . . .	28
5.5	GedGNN . . . . .	29
5.6	Encountered Gaps . . . . .	30
<b>6</b>	<b>Methodology and Experimentation</b>	<b>31</b>
<b>7</b>	<b>Discussion and Conclusions</b>	<b>31</b>

# 1 Introduction

Graphs are fundamental structures in computer science and mathematics that model relationships between entities. They consist of vertices (or nodes) and edges (or links) that connect pairs of vertices. This simple yet powerful representation can capture a wide variety of real-world scenarios, providing a versatile tool for analyzing complex systems. For instance, social networks can be represented as graphs where nodes denote individuals, and edges represent their connections or interactions, enabling the study of social dynamics, influence, and community formation. In biology, protein-protein interaction networks, neural networks of the brain, and ecological networks can all be modeled as graphs, facilitating the understanding of biological processes, brain functionality, and ecosystem interdependencies. Similarly, in transportation, cities can be nodes and roads or flights can be edges, creating a network that facilitates route optimization, urban planning, and logistical efficiency. Graphs can also model communication networks, where devices are nodes and connections are edges, allowing for analysis of data flow, network robustness, and optimization of resource allocation.

Graph theory provides a rich framework for analyzing these structures, with properties such as connectivity, centrality, and clustering coefficient helping to understand the underlying patterns and behaviors within the network. Connectivity measures how well the nodes are connected, centrality identifies the most important nodes within the graph, and clustering coefficient gives insight into the degree to which nodes tend to cluster together. Additionally, other important graph properties include graph diameter, which measures the longest shortest path between any two nodes, and graph density, which indicates the level of interconnectedness in the network. These properties help in uncovering critical information about the structure and function of the graph, enabling more effective analysis and decision-making.

The Graph Edit Distance (GED) problem is a critical measure in graph theory, providing a similarity metric between two graphs. GED quantifies how many operations (such as insertions, deletions, and substitutions of nodes and edges) are required to transform one graph into another. This measure is invaluable for various applications, including bioinformatics, where it can compare molecular structures to identify potential drug candidates or understand evolutionary relationships. In computer vision, GED is crucial for object recognition, where the structural similarity between graphical representations of different objects must be assessed to identify and classify them accurately. Other graph similarity measures include graph isomorphism, which checks for exact structural similarity, and subgraph isomorphism, which identifies if one graph is a subgraph of another, useful for pattern matching and searching within larger networks.

Knowing the exact GED between two graphs can provide profound insights. For example, in bioinformatics, understanding the similarity between different molecular structures can lead to the discovery of new drugs and therapeutic targets by revealing structural patterns that correlate with biological activity.

In social network analysis, GED can help detect communities or clusters of users with similar interaction patterns, aiding in the identification of influential individuals, the spread of information, or the formation of social groups. Moreover, in pattern recognition and image analysis, GED can assist in identifying objects and understanding their structural relationships, enhancing the accuracy and reliability of automated systems. The ability to quantify the similarity between graphs allows for more precise and meaningful comparisons, driving innovations and improvements across these fields.

However, computing the exact GED is notoriously difficult due to its high computational complexity. The problem is NP-hard, meaning that the time required to solve it grows exponentially with the size of the graphs, making it computationally prohibitive for large graphs. This involves an exhaustive search over all possible edit paths, which is impractical for real-world applications. Various heuristics and approximation algorithms have been proposed, but they often struggle to balance accuracy and computational efficiency, leading to trade-offs that can impact the reliability of the results. The NP-hard class encompasses problems that are at least as hard as the hardest problems in NP, and no known polynomial-time algorithm can solve them. This inherent difficulty underscores the challenge of computing GED and the necessity for developing efficient approximation methods that can provide accurate results within reasonable timeframes.

Neural networks, a cornerstone of modern machine learning, have revolutionized numerous fields by providing robust methods for handling complex, high-dimensional data. A neural network is a series of algorithms that attempt to recognize underlying relationships in a set of data through a process that mimics the way the human brain operates. They consist of interconnected layers of nodes, or neurons, each capable of processing inputs and producing outputs. Neural networks are trained using large datasets where they adjust their internal parameters based on the error between the predicted outputs and the actual outputs. This training process involves forward propagation, where inputs are passed through the network to generate outputs, and backpropagation, where the error is propagated back through the network to update the weights, thereby improving the model’s accuracy. Neural networks have been successfully applied to a wide range of tasks, including image and speech recognition, natural language processing, and more recently, graph-structured data analysis, demonstrating their versatility and effectiveness.

Graph Neural Networks (GNNs) are a specialized type of neural network designed to work directly with graph-structured data. GNNs aim to leverage the graph’s inherent structure by performing convolution operations over the nodes and edges, capturing both local and global graph properties. This makes them well-suited for various tasks, including node classification, link prediction, and graph classification. Given their ability to learn complex patterns and representations, GNNs hold promise for approximating the GED. GNNs operate by iteratively updating the representation of each node based on its neighbors, effectively capturing the dependencies and relationships within the graph. This iterative process enables GNNs to learn hierarchical representations that are cru-

cial for understanding and analyzing graph-structured data, allowing for more accurate and meaningful predictions in various applications.

This thesis reviews a range of key articles to explore the current state-of-the-art methods in GED computation, encompassing both neural network-based approaches and traditional methods. The reviewed works span various innovative strategies, each attempting to tackle the challenges of GED computation from different angles. By critically analyzing these methods, this review aims to identify their strengths and limitations, offering insights into potential improvements. The seminal paper on SimGNN [3] serves as a foundation for many subsequent works, introducing a neural network-based approach to GED computation that has inspired numerous advancements. More recent works, such as GedGNN [12], continue to push the boundaries of what is possible, integrating novel techniques and improving upon previous methods.

Improving GED computation methods is crucial for enhancing the performance of numerous applications that rely on graph similarity measures. For instance, more efficient and accurate GED computation can lead to breakthroughs in drug discovery by enabling faster and more precise comparison of molecular structures, facilitating the identification of new compounds with therapeutic potential. In social network analysis, it can facilitate the detection of more accurate community structures, leading to better understanding and management of social dynamics, improving the effectiveness of interventions and policy decisions. In computer vision, improved methods can enhance object recognition systems, making them more reliable and efficient, which is vital for applications ranging from autonomous vehicles to security systems. The implications of better GED computation extend to numerous domains, highlighting the importance of continued research and development in this area to unlock new possibilities and advancements.

As we delve into the review of these articles, the goal is to provide a comprehensive overview of the advancements in GED computation. By highlighting innovative strategies and pinpointing areas for further research, this thesis aims to contribute to the ongoing efforts to refine and enhance GED computation methods. We will reproduce the results of key recent papers, such as the one proposing GedGNN, to validate their findings. Additionally, this thesis will offer critical and constructive advice on aspects such as code quality, the fairness of presented results, and the limitations of the datasets used. We will discuss issues like poor dataset quality and propose solutions, including artificial dataset generation and the development of neural networks that can be tested on any dataset, ensuring a fairer evaluation. This comprehensive review aims to bridge the gap between existing methods and the potential for new, more effective techniques, ultimately contributing to the broader field of graph theory and its myriad applications. By providing a thorough analysis and constructive feedback, this thesis seeks to guide future research and development efforts, paving the way for advancements that will enhance the accuracy, efficiency, and applicability of GED computation methods in various fields.

## 2 Graph Data Structure

A *graph*  $G$  [Figure 1] is a non-linear data structure consisting of vertices and edges. The vertices are sometimes also referred to as nodes and the edges are arcs that connect any two nodes in the graph. Graphs are used to represent relationships between different entities and have applications in many fields including Computer Science, Physics, Biology, Chemistry, Optimization Theory, Social Sciences, and many more. They are fundamental in modeling networks such as social networks, communication networks, biological networks, and transportation systems, making them indispensable tools for analyzing and solving complex problems. In everyday tasks, graphs can represent relationships in recommendation systems, routing algorithms in GPS navigation, workflow optimization, and resource allocation in various industries.

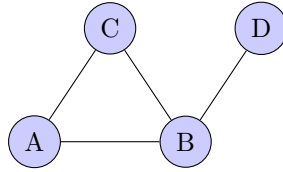


Figure 1: Simple undirected graph example.

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

- $V$  is a finite set of vertices (or nodes). Each vertex represents an entity or a data point, and the set of vertices  $V$  is often denoted as  $V = \{v_1, v_2, \dots, v_n\}$  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,  $E \subseteq \{\{u, v\} \mid u, v \in V \text{ and } u \neq v\}$ . Each edge signifies a relationship or connection between the pair of vertices it links.

For instance The 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

Graphs can be classified into various types based on their properties, including:

- **Directed Graph** [Figure 2]: A directed graph (or digraph) is a graph in which every edge has a direction, represented as an ordered pair  $(u, v)$  where  $u, v \in V$  and  $u \neq v$ . It is used in various applications such as web page ranking, where links from one page to another have a specific direction, and citation networks, where one paper cites another.

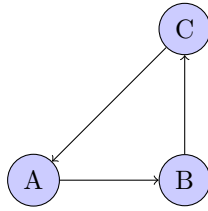


Figure 2: Directed graph where edges have directions indicated by arrows.

- **Undirected Graph** [Figure 3]: An undirected graph is a graph in which the edges do not have a direction, represented as an unordered pair  $\{u, v\}$  where  $u, v \in V$  and  $u \neq v$ . This type of graph is commonly used to model social networks where the connections (friendships) are mutual, indicating that if one person is friends with another, the reverse is also true.

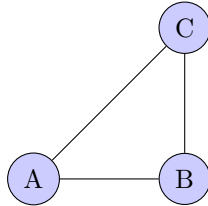


Figure 3: Undirected graph example where edges are bidirectional (there are no arrows).

- **Weighted Graph** [Figure 4]: A weighted graph is a graph in which each edge has an associated weight or cost, represented as a function  $w : E \rightarrow \mathbb{R}$  where  $w(e)$  is the weight of edge  $e \in E$ . This is particularly useful in transportation networks where the weights can represent distances, travel times, or costs associated with traveling between locations.

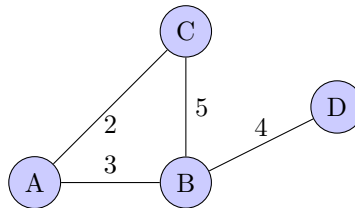


Figure 4: Weighted graph example where each edge is labeled with a weight.

- **Simple Graph** [Figure 5]: A simple graph is a graph that has no loops (edges connecting a vertex to itself) and no multiple edges (more than one edge connecting the same pair of vertices). Simple graphs are the most basic type of graph, with straightforward structures that make them easy

to analyze. They are used in many basic network models to simplify the analysis and understand the fundamental properties of the network.

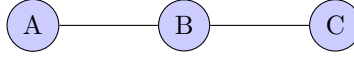


Figure 5: Simple graph example with a linear connection between vertices A, B, and C, with no loops or multiple edges.

- **Complete Graph** [Figure 6]: A complete graph is a graph in which there is exactly one edge between each pair of distinct vertices. Formally, a complete graph on  $n$  vertices, denoted as  $K_n$ , has  $E = \{\{u, v\} \mid u, v \in V, u \neq v\}$ . Complete graphs are used in scenarios where maximum connectivity is required, such as in certain network topologies and in combinatorial optimization problems.

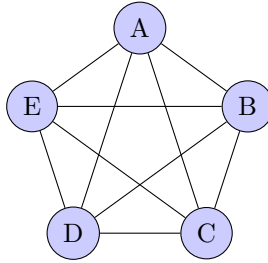


Figure 6: Complete graph example where each node is connected to each other.

- **Bipartite Graph** [Figure 7]: A bipartite graph is a graph whose vertices can be divided into two disjoint sets  $U$  and  $W$  such that every edge connects a vertex in  $U$  to a vertex in  $W$ . Bipartite graphs are used to model relationships between two different classes of objects. For example, in job assignments, vertices in  $U$  can represent jobs, and vertices in  $W$  can represent workers, with edges indicating which workers are assigned to which jobs.

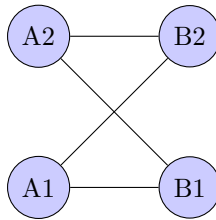


Figure 7: Bipartite graph example with two distinct sets of vertices with edges connecting vertices across the sets but not within them.



- **Multigraph** [Figure 8]: A multigraph is a graph which allows multiple edges between the same pair of vertices. Formally,  $G = (V, E)$  where  $E$  is a multiset of unordered pairs of vertices. Multigraphs are used to model scenarios where multiple relationships or interactions can exist between entities. For example, in transportation networks, multiple routes or connections can exist between the same pair of locations, and these multiple edges can represent different routes or modes of transport.

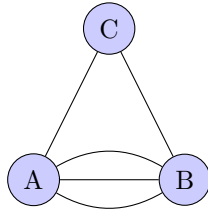


Figure 8: Multigraph example with multiple edges between vertices A and B.

- **Cyclic Graph** [Figure 9]: A cyclic graph is a graph that contains at least one cycle, where a cycle is a path of edges and vertices wherein a vertex is reachable from itself. Cyclic graphs are used to model processes or systems where feedback loops are present. For example, in certain biological systems or in recurrent neural networks, cycles can represent the feedback mechanisms or recurrent connections.

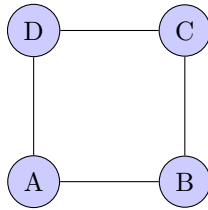


Figure 9: Cyclic graph example with a cycle A-B-C-D-A.

- **Acyclic Graph** [Figure 10]: An acyclic graph is a graph with no cycles. A directed acyclic graph (DAG) is a directed graph with no directed cycles. Acyclic graphs, especially directed acyclic graphs (DAGs), are used in scenarios such as scheduling tasks, where dependencies must not form cycles. In such cases, a task can only start once all its prerequisite tasks are completed, and the absence of cycles ensures that there are no circular dependencies.

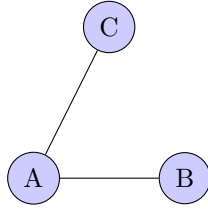


Figure 10: Acyclic graph example with no cycles.

## 2.2 Graph Representation

Graphs can be represented in various ways, including:

- **Adjacency Matrix** [Figure 11]: An adjacency matrix  $A$  for a graph  $G = (V, E)$  is a square matrix of size  $|V| \times |V|$ . The entry  $A_{ij}$  is 1 if there is an edge between vertices  $v_i$  and  $v_j$ , and 0 otherwise. This representation is particularly useful for dense graphs, where the number of edges is close to the maximum possible number of edges. It allows for efficient querying of edge existence and is easy to implement for algorithms that require frequent checks of edge presence. However, the space complexity is  $O(|V|^2)$ , which can be prohibitive for large graphs with many vertices.

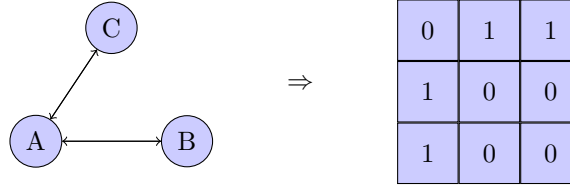
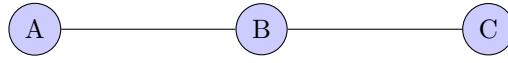


Figure 11: Adjacency Matrix example.

- **Adjacency List** [Figure 12]: An adjacency list is a collection of lists or arrays where each list corresponds to a vertex and contains all the vertices adjacent to that vertex. For a graph  $G = (V, E)$ , the adjacency list can be represented as an array of lists  $\{L_1, L_2, \dots, L_n\}$ , where each  $L_i$  contains the neighbors of vertex  $v_i$ . This representation is more space-efficient for sparse graphs, where the number of edges is much smaller than the number of possible edges. It facilitates efficient traversal operations such as breadth-first search (BFS) and depth-first search (DFS), where only the relevant neighbors of each vertex need to be examined.



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

Figure 12: Adjacency List example.

## 2.3 Properties of Graphs

Graphs possess various properties that help in their analysis and application. Some of these properties include:

- **Degree** [Figure 13]: The degree of a vertex is the number of edges incident to it. For a vertex  $v$  in a graph  $G = (V, E)$ , the degree  $\deg(v)$  is the count of edges connected to  $v$ . In directed graphs, the in-degree represents the number of incoming edges and the out-degree represents the number of outgoing edges. High-degree vertices often play a crucial role in the graph, indicating significant or highly connected nodes.

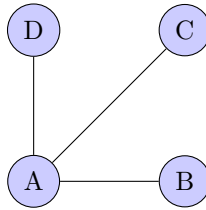


Figure 13: Degree example showing vertex A with degree 3.

- **Connectivity** [Figure 14]: Connectivity refers to how well nodes are interconnected within a graph. A graph is said to be connected if there is a path between every pair of vertices. This property is vital for understanding network reliability and robustness, as it ensures that all nodes can communicate or reach each other directly or indirectly.

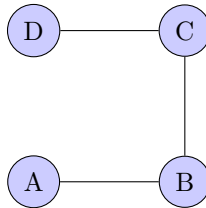


Figure 14: Connectivity example showing a connected graph.

- **Centrality** [Figure 15]: Centrality measures are used to identify the most important vertices within a graph. Different types of centrality include

degree centrality, which counts the number of direct connections a vertex has; closeness centrality, which measures how quickly a vertex can access other vertices; and betweenness centrality, which quantifies how often a vertex acts as a bridge along the shortest path between other vertices. These measures provide various insights into the roles and influence of vertices in a network.

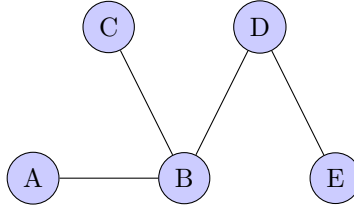


Figure 15: Centrality example showing vertex B as a central node with high degree centrality.

- **Clustering Coefficient** [Figure 16]: The clustering coefficient of a vertex measures the extent to which neighbors of the vertex are also connected to each other. A high clustering coefficient indicates a tightly-knit community within the graph. In mathematical terms, it is calculated as the ratio of the number of actual edges to the number of possible edges among the neighbors of a vertex.

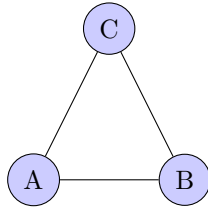


Figure 16: Clustering coefficient example showing a triangle, indicating a high clustering coefficient.

- **Graph Diameter** [Figure 17]: The diameter of a graph is the length of the longest shortest path between any pair of vertices. This metric provides an indication of the "spread" of the graph and helps in understanding how far apart vertices can be in terms of the shortest path distance.



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

- **Graph Density** [Figure 18]: Graph density is defined as the ratio of the number of edges in the graph to the number of possible edges between vertices. For a graph with  $n$  vertices, the maximum number of edges is  $\frac{n(n-1)}{2}$  in an undirected graph. Density provides a measure of how close the graph is to being a complete graph, where all possible edges are present.

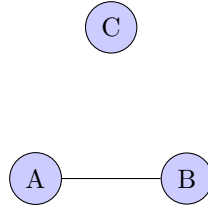


Figure 18: Graph density example showing a sparse graph with few edges relative to the number of vertices.

### 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 in a sort of magical way.

#### 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 it self. There is a plenitude of way to organize models, but the simplest we can think of [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 we are working with 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 we only have one hidden layer, but as we will see 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 if we are building a regression task model the output

layer could consist 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:

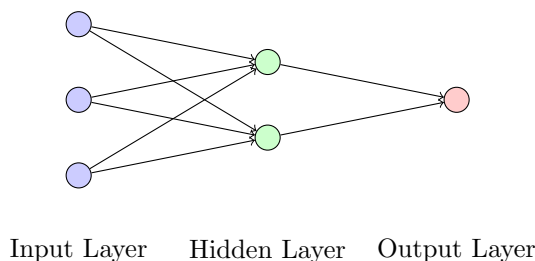


Figure 19: A simple neural network with one hidden layer.

### 3.1.1 Activation Functions

As already said, 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, we could build 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

In supervised learning data used is said to *labeled* meaning that for each *example* its *class* is known. Unsupervised and Semi-supervised learning also exist but we are going to deal with these. 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 *gradient descent*. 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 and so on and so forth. 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 space and the best one are then selected. It's especially important to prevent a phenomenon known as *overfitting*.
- **Test Set:** This small to medium sized subset is used to evaluate the model's performance on data it has not seen before. By testing the model on never seen data we obtain unbiased measures metrics 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 overfitting, techniques such as regularization, dropout, and early stopping are employed to ensure the model generalizes well to unseen data. The end goal is to have a model that generalize well with the 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 maximize. 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:

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

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:

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

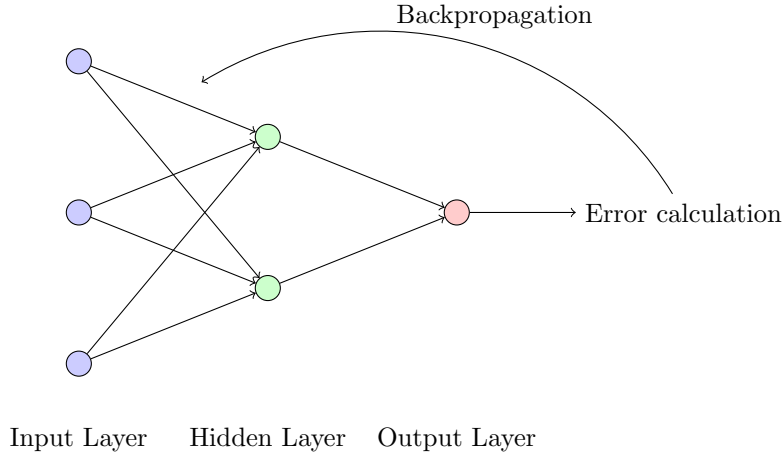


Figure 20: Backpropagation in action within a simple neural network.



### 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  $\eta$  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.

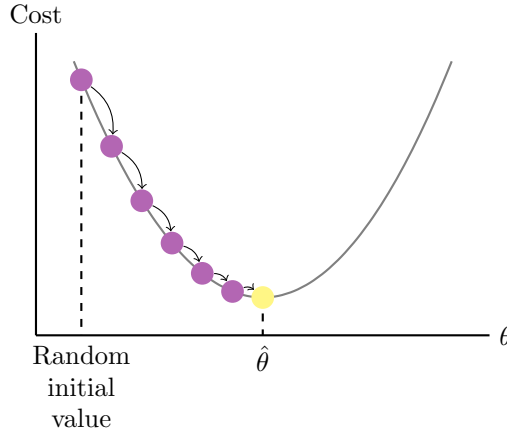


Figure 21: Gradient descent method applied on a concave function.

## 3.3 Advanced Topics in Neural Networks

### 3.3.1 Deep Neural Networks

*Deep neural networks* (DNN) [Figure 22] differ from simple one 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.

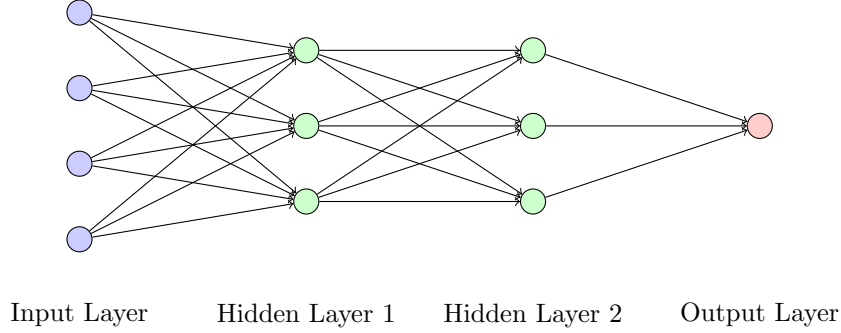


Figure 22: A Deep neural network with two hidden layers.

### 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.

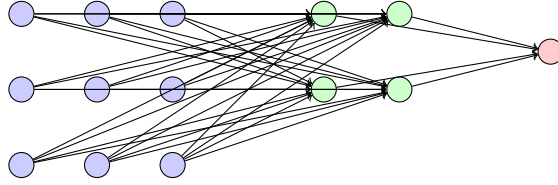


Figure 23: A simple convolutional neural network architecture.

### 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  $\sigma$  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.

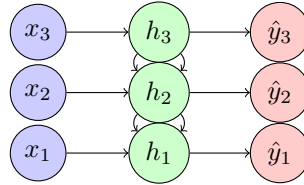


Figure 24: Recurrent Neural Network (RNN) unrolled through time.

### 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 were first introduced in 2017 with the paper *Attention is all you need* [16]. Let's say we are dealing with text, then each word in the input text is associated with a *key* and the element we are focusing on 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 for 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  $\{k_1, k_2, \dots, k_n\}$  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 pretty much be integrated with any type of neural network even though that's not always necessary.

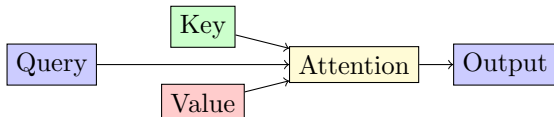


Figure 25: Attention mechanism in neural networks.

### 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  $\sigma$  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).

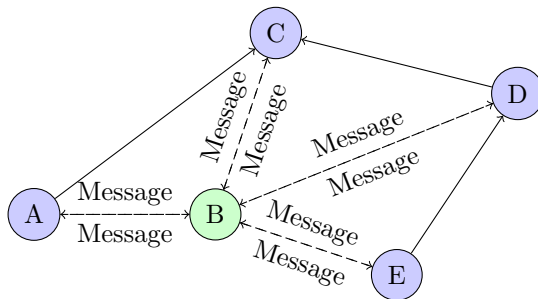


Figure 26: Message passing between node B and its neighbors in a GNN.

We are specifically going to focus on GNNs since every state of the art model trying to address the graph edit distance problem [section 4] which make use of artificial neural networks do 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 properties of pair 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, we are going to explore 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 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 solely 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  $f : V_1 \rightarrow V_2$  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.

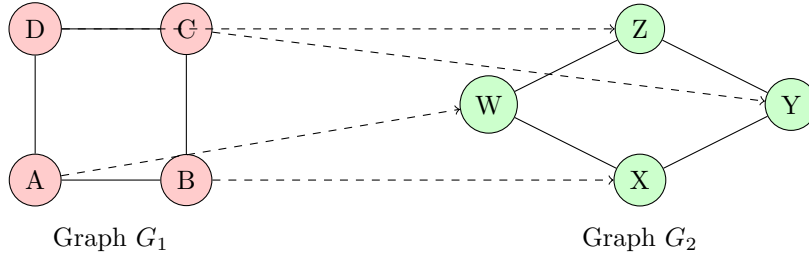


Figure 27: Graph Isomorphism:  $G_1$  (Square) and  $G_2$  (Rhombus).

## 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 classifiers. However, they can be computationally intensive if not carefully handled and also require careful tuning of parameters.

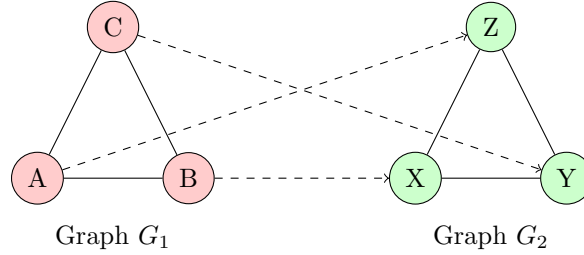


Figure 28: Graph Kernels: Example Graphs with Similar Structures.

### 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). GED 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  $\Sigma$  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  $\Sigma^*$  denotes the set of all finite sequences of operations from  $\Sigma$ , and  $o$  represents an individual operation within a sequence  $\sigma$ .

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.

#### 4.3.1 Atomic Operations

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]:

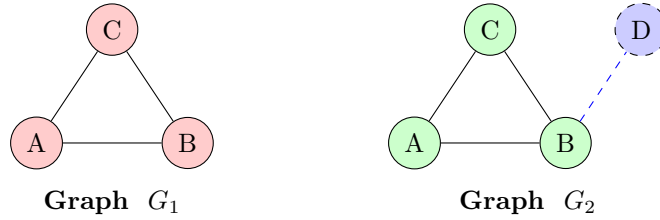


Figure 29: Graph Edit Distance: Transforming  $G_1$  to  $G_2$  by adding vertex  $D$  and edge  $(B,D)$ .

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 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).

The timeline we are going to explore starts with *SimGNN* in 2019 [3], the first prominent approach to utilize neural networks for computing GED. After that many other models have been built mostly on SimGNN's foundation each



time trying to perform a little better. Finally the timeline ends with most recent and promising work introducing *GedGNN* [12]. All these models try to effectively predict the GED between two graphs, but as we will see, everything seems still to be in early stage of development.

## 5.1 Timeline

2019, *SimGNN: A Neural Network Approach to Fast Graph Similarity Computation* [3]: Introduces SimGNN, addressing graph similarity computation using neural networks. It features a learnable embedding function, an attention mechanism to focus on important nodes, and a pairwise node comparison method, achieving better generalization and computational efficiency compared to baselines.

2020, *Learning Graph Edit Distance by Graph Neural Networks* [14]: Introduces a framework combining deep metric learning with traditional approximations of graph edit distance using geometric deep learning. The approach employs a message passing neural network (MPNN) to capture graph structure and compute graph distances efficiently, showing superior performance in graph retrieval and competitive results in graph similarity learning.

2020, *Combinatorial Learning of Graph Edit Distance via Dynamic Embedding* [18]: Introduces a hybrid approach for solving the GED problem by integrating a dynamic graph embedding network with an edit path search procedure, enhancing interpretability and cost-efficiency. The learning-based A\* algorithm reduces search tree size and saves time with minimal accuracy loss.

2021, *Graph Partitioning and Graph Neural Network-Based Hierarchical Graph Matching for Graph Similarity Computation* [19]: Introduces PSimGNN, which partitions input graphs into subgraphs to extract local structural features, then uses a novel GNN with attention to map subgraphs to embeddings, combining coarse-grained interaction among subgraphs with fine-grained node-level comparison to predict similarity scores.

2021, *Noah: Neural Optimized A\* Search Algorithm for Graph Edit Distance Computation* [20]: Introduces Noah, combining A\* search algorithm and Graph Path Networks (GPN) for approximate GED computation. Noah learns an estimated cost function using GPN, incorporates pre-training with attention-based information, and adapts an elastic beam size to reduce search complexity.

2021, *Learning Efficient Hash Codes for Fast Graph-Based Data Similarity Retrieval* [17]: Introduces HGNN (Hash Graph Neural Network), a model designed for efficient graph-based data retrieval by leveraging GNNs and hash learning algorithms. HGNN learns a similarity-preserving graph representation and computes compact hash codes for fast retrieval and classification tasks.

2021, *More Interpretable Graph Similarity Computation via Maximum Common Subgraph Inference* [7]: Introduces INFMCS, an interpretable end-to-end paradigm for graph similarity learning, leveraging the correlation between similarity score and Maximum Common Subgraph (MCS), combining transformer encoder layers with graph convolution for superior performance and interpretability.

2021, *H2MN: Graph Similarity Learning with Hierarchical Hypergraph Matching Networks* [21]: Introduces H2MN, which measures similarities between graph-structured objects by transforming graphs into hypergraphs and performing subgraph matching at the hyperedge level, followed by a multi-perspective cross-graph matching layer.

2022, *TaGSim: Type-aware Graph Similarity Learning and Computation* [2]: Proposes TaGSim, a framework that addresses the limitations of traditional GED methods by incorporating type-specific graph edit operations. TaGSim models the transformative impacts of different graph edits (node and edge insertions, deletions, and relabelings) separately, creating type-aware embeddings and using these embeddings for accurate GED estimation. The framework demonstrates superior performance on real-world datasets compared to existing GED solutions.

2023, *Efficient Graph Edit Distance Computation Using Isomorphic Vertices* [6]: Proposes a novel approach for reducing the search space of GED computation by leveraging isomorphic vertices, targeting redundant vertex mappings and significantly cutting computation costs for exact GED.

2023, *Exploring Attention Mechanism for Graph Similarity Learning* [15]: Proposes a unified framework with attention mechanisms, combining graph convolution and self-attention for node embedding, cross-graph co-attention for interaction modeling, and graph similarity matrix learning for score prediction, showing superior performance on benchmark datasets.

2023, *Graph Edit Distance Learning via Different Attention* [10]: Introduces DiffAtt, a novel graph-level fusion module for GNNs to compute GED efficiently by leveraging graph structural differences using attention mechanisms, incorporated into the GSC model REDRAFT, achieving state-of-the-art performance on benchmark datasets.

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

2023, *GREED: A Neural Framework for Learning Graph Distance Functions* [13]: Introduces GREED, a siamese GNN designed to learn GED and Subgraph Edit Distance (SED) in a property-preserving manner, achieving superior accuracy and efficiency compared to state-of-the-art methods.

2023, *MATA\*: Combining Learnable Node Matching with A\* Algorithm for Approximate Graph Edit Distance* [9]: Introduces MATA\*, a hybrid approach for approximate GED computation leveraging GNNs and A\* algorithms, focusing on learning to match nodes rather than directly regressing GED.

2023, *Multilevel Graph Matching Networks for Deep Graph Similarity Learning* [8]: Proposes MGMN, a multilevel graph matching network capturing cross-level interactions, comprising a Node-Graph Matching Network (NGMN) and a siamese GNN for global-level interactions, demonstrating superior performance as graph sizes increase.

2023, *Wasserstein Graph Distance Based on L1-Approximated Tree Edit Dis-*

*tance Between Weisfeiler-Lehman Subtrees* [4]: Proposes the WWLS distance, combining WL subtrees with L1-approximated tree edit distance (L1-TED), capable of detecting subtle structural variations in graphs, demonstrating superiority in metric validation and graph classification tasks.

2023, *Computing Graph Edit Distance via Neural Graph Matching* [12]: Introduces GEDGNN, a deep learning framework for computing GED by focusing on graph conversion rather than GED value prediction alone. GEDGNN predicts GED values and a matching matrix, followed by a post-processing algorithm for extracting high-quality node matchings.

## 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 at 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.

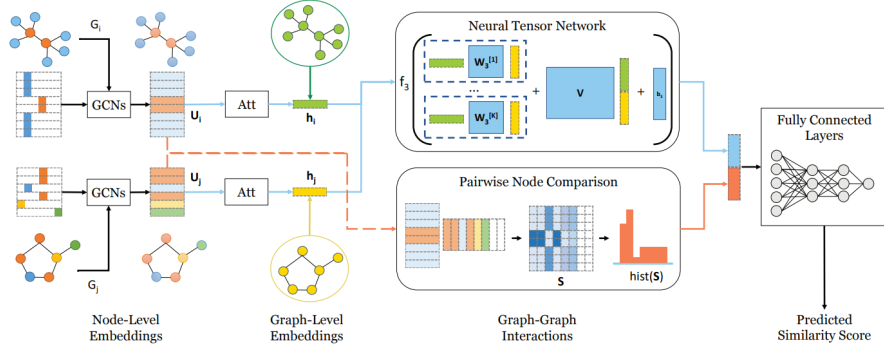


Figure 30: SimGNN architecture overview.

### 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.

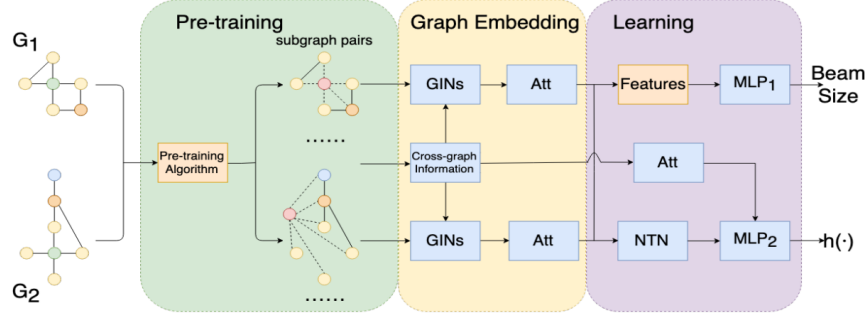


Figure 31: GPN architecture overview.

## 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 reevaluated and it is now thought as the summation of three different values: *ged<sub>nc</sub>* the number of node relabelling, *ged<sub>in</sub>* the number of node insertions/deletions, *ged<sub>ie</sub>* 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.

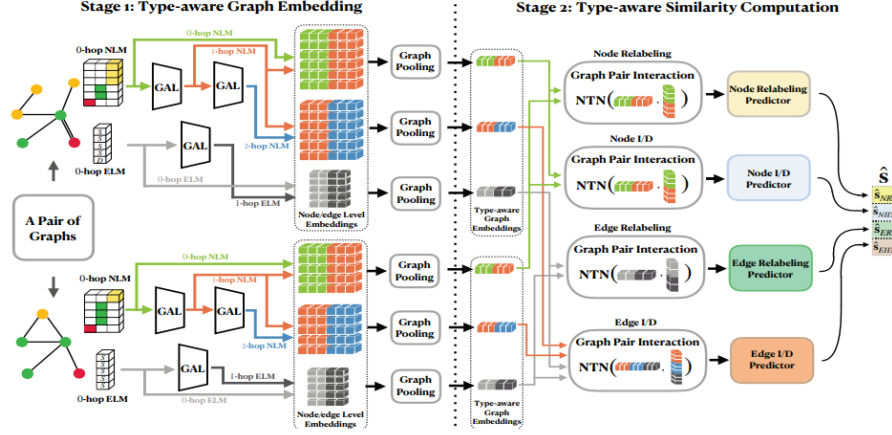


Figure 32: TaGSim architecture overview.

## 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.

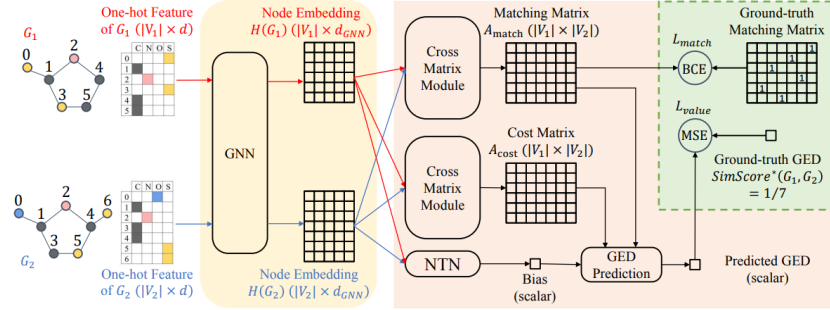


Figure 33: GedGNN architecture overview.

## 5.6 Encountered Gaps

When studying GedGNN’s `codebase` a lot of problems and concerns have become evident. Perhaps most of these gaps are due to the intrinsic nature of copy pasting from paper to paper over time and to the lack of researcher’s ability to extensively testing their work.

A primary issue is the quality of the codebase itself, there is a lack of standardization and employments of best practices, lots of similar methods that all seem to do the same thing, lots of classes, lots of confusion, lots of unhandled errors. Code almost seem hardcoded here and there, not allowing for customization and manipulation. The code is not easy to read and the documentation, although written, is very poor.

Strictly correlated to code quality and testing scenarios is the scalability issues 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 feasible on CPUs but as soon as a bigger dataset are used for test-training new models, problems have become evident with long training times per epoch.

It is not clear why the codebase as is does restrict the usage of graphs for those with only 10 nodes or less in both training, testing and validation scenarios. Perhaps due to the presence of many imperative algorithm with high computational time such as the k-best post processing algorithm that retrieves edit paths; also code as is does break (at least for me) when testing on the IMDB dataset (with any model). Additionally, there is no use of artificial dataset generation, despite being present in the code, creating more confusion and several if statements that will never be reached.

Data used for training is also an issue persisting from SimGNN’s times. Datasets used are always the same but the issue (which is not an issue but rather a constructive critic), is that approximate GEDs are used as labels instead of real ones. Datasets are mostly small (with graphs with less than 10 nodes in general) and with modern hardware GED wouldn’t take much time to compute exactly. Also, in almost any codebase seen till date, training is structured in such

a way that for each pair of graph present in the training set the corresponding GED is required.

But the most important thing that has been shown up is that testing has not been performed in a fair way. It is not clear why, but when dealing with the codebase as is, it is not possible to test a model that has been trained on a specific dataset on a different dataset. Perhaps to not show the evident lack of generalization emerged after doing modifications to the code to perform fair testing [section 6].

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

## 7 Discussion and Conclusions

## 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] Jiayi Lv, Liang Zhang, Yi Huang, Jiancheng Huang, and Shifeng Chen. Graph edit distance learning via different attention. 2023.
  - [11] Bernhard Olkoph. 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 Chakravarthy, 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.