Master Thesis

Federico Calabrò 247976

Department of Mathematics and Computer Science
2024

Contents

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 [2] 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 [10], 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.

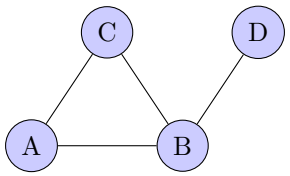


Figure 1: Simple undirected graph example.

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 $V = \{v_1, v_2, \ldots, v_n\}$ 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, $E \subseteq \{\{u, v\} \mid u, v \in V \text{ and } u \neq v\}$. 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.

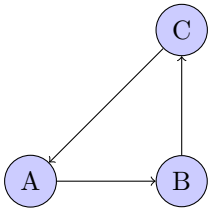


Figure 2: Directed graph where edges have directions indicated by arrows.

- **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.

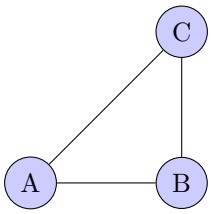


Figure 3: Undirected graph example where edges are bidirectional (there are no arrows).

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

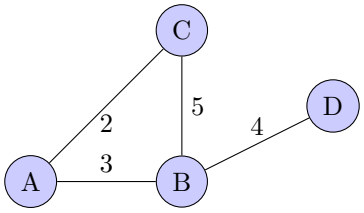


Figure 4: Weighted graph example where each edge is labeled with a weight.

- **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.

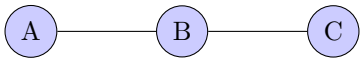


Figure 5: Simple graph example with a linear connection between vertices A, B, and C, with no loops or multiple edges.

- **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.

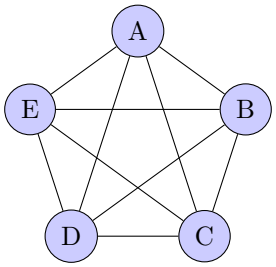


Figure 6: Complete graph example where each node is connected to each other.

- **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.

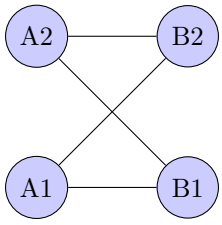


Figure 7: Bipartite graph example with two distinct sets of vertices with edges connecting vertices across the sets but not within them.

- **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.

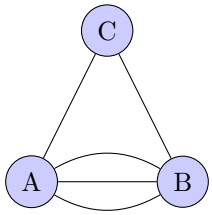


Figure 8: Multigraph example with multiple edges between vertices A and B.

- **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.

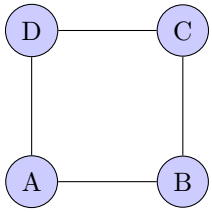


Figure 9: Cyclic graph example with a cycle A-B-C-D-A.

- **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.

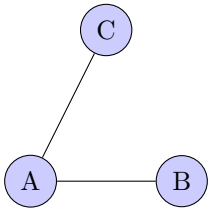


Figure 10: Acyclic graph example with no cycles.

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.

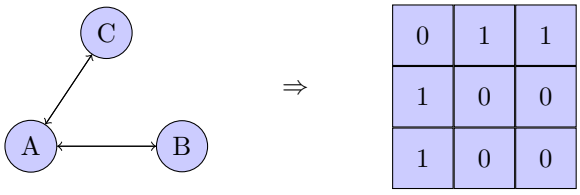
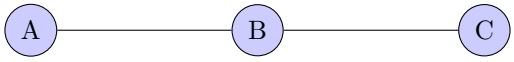


Figure 11: Adjacency Matrix example.

- **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 $\{L_1, L_2, \ldots, L_n\}$, 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.



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

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.

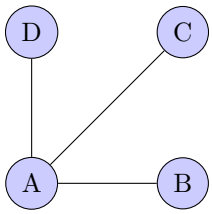


Figure 13: Degree example showing vertex A with degree 3.

- **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.

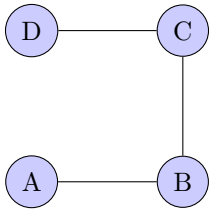


Figure 14: Connectivity example showing a connected graph.

- **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.

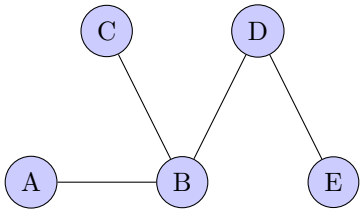


Figure 15: Centrality example showing vertex B as a central node with high degree centrality.

- **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.

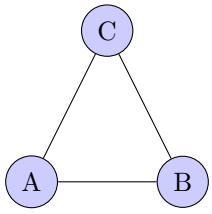


Figure 16: Clustering coefficient example showing a triangle, indicating a high clustering coefficient.

- **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.

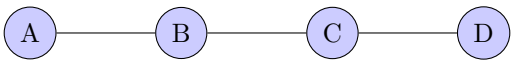


Figure 17: Graph diameter example showing the longest shortest path A-B-C-D with diameter 3.

- **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.

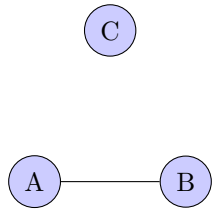


Figure 18: Graph density example showing a sparse graph with few edges relative to the number of vertices.

3 Neural Networks

A *neural network* is a sophisticated computational model inspired by the intricate and highly parallel nature of biological neural networks found in the human brain. These models are designed to recognize patterns and solve complex problems by emulating the way neurons interact through synaptic connections. Just as the human brain processes information through a vast network of interconnected neurons, artificial neural networks utilize a collection of interconnected units, called neurons or nodes, to process data. Each neuron performs a simple computation, but when combined in a network, they can tackle a wide range of tasks from simple pattern recognition to advanced decision-making processes.

3.1 Basic Structure of a Neural Network

A simple neural network [Figure 19] consists of three main types of layers that collaborate to transform input data into meaningful outputs:

- **Input Layer**: This layer consists of input neurons that receive the initial data. Each neuron in this layer corresponds to a feature in the input dataset. For example, in image recognition, each neuron might represent a pixel value.
- **Hidden Layers**: These intermediate layers are where the actual computation and learning occur. Hidden layers can be one or more in number, depending on the complexity of the network. Each hidden layer consists of neurons that apply weights and activation functions to the inputs received from the previous layer, thereby transforming the data progressively.

- **Output Layer**: The final layer in the network, which produces the output. The number of neurons in the output layer corresponds to the number of possible classes or output values in the problem. For instance, in a binary classification problem, there might be a single neuron that outputs a probability.

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

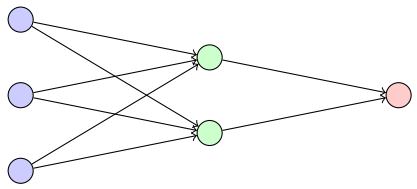


Figure 19: A simple neural network with one hidden layer.

3.1.1 Activation Functions

In a neural network, each neuron from a previous layer connects to one or more neurons in subsequent layers through activation functions. These functions introduce non-linearity into the model, enabling it to capture complex patterns and relationships within the data. Without activation functions, a neural network, regardless of its depth, would only perform linear transformations, limiting its capacity to solve more intricate problems. Some commonly used activation functions include:

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

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

- **Tanh**: The tanh function maps real-valued numbers 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 is the most commonly used activation function in modern neural networks. It outputs the input directly if it is positive; otherwise, it outputs zero.

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

3.2 Training Neural Networks

Training a neural network is an iterative process aimed at finding the optimal weights that minimize the error between the network's predictions and the actual target values. This process involves adjusting the weights of the connections through a technique known as *backpropagation* and using an optimization algorithm such as *gradient descent*. To effectively train a model, the dataset is typically divided into three distinct subsets:

- **Training Set**: This subset is used to adjust the weights of the network. The model learns and updates its weights based on this data to minimize the error between its predictions and the actual values.
- **Validation Set**: This subset is used to tune *hyperparameters*, which are the parameters set before the training process begins (such as the learning rate, number of hidden layers, and number of neurons in each layer). Hyperparameters influence the model's architecture and training process, and finding the right values is crucial for achieving optimal performance. The validation set helps in selecting these hyperparameters and in monitoring the model's performance to prevent *overfitting*.
- **Test Set**: This subset is used to evaluate the model's performance on data it has not seen before. It provides an unbiased assessment of how well the model generalizes to new, unseen data.

Overfitting occurs when a model learns the training data too well, capturing noise and details that do not generalize to new data. This results in high accuracy on the training set but poor performance on the test set. To mitigate overfitting, techniques such as regularization, dropout, and early stopping are employed to ensure the model generalizes well to unseen data.

3.2.1 Backpropagation

Backpropagation [Figure 20] is a fundamental algorithm used to compute the gradient of the loss function with respect to each weight in the network by applying the chain rule. This algorithm operates in two main phases: the forward pass and the backward pass. During the forward pass, input data is propagated through the network layer by layer until the final output is produced. The error is then calculated using a loss function, such as mean squared error for regression tasks. In the backward pass, this error is propagated back through the network, layer by layer, allowing the algorithm to compute the gradients of the loss function with respect to each weight. These gradients indicate the direction in which each weight should be adjusted to minimize the error, guiding the weight update process.

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:

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

In the context of neural networks, the chain rule helps in propagating the error gradient back through the network, enabling the adjustment of weights based on their contribution to the total error.

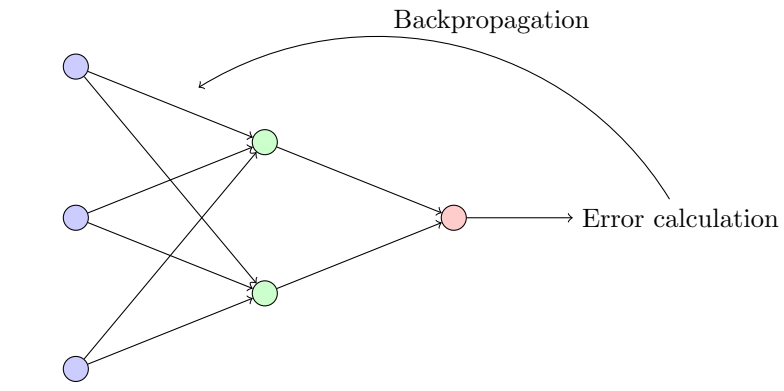


Figure 20: A simple neural network with one hidden layer.

3.2.2 Gradient Descent

Gradient descent [Figure 21] is an optimization algorithm used to minimize the loss function by iteratively adjusting the weights in the direction that reduces the error the most. At each iteration, the algorithm updates the weights by moving them in the opposite direction of the gradient of the loss function with respect to the weights. The size of these steps is determined by the learning rate, a crucial hyperparameter that needs to be carefully chosen to ensure the algorithm converges efficiently. The weight update rule for a weight w can be expressed as:

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

where η is the learning rate. Various variants of gradient descent, such as stochastic gradient descent (SGD) and mini-batch gradient descent, offer different trade-offs between computation time and convergence stability. Additionally, techniques like momentum, RMSprop, and Adam can further enhance the optimization process by adapting the learning rate during training. Momentum, for example, helps accelerate SGD in the relevant direction and dampens oscillations. RMSprop and Adam adapt the learning rate for each parameter, making the optimization more efficient and robust.

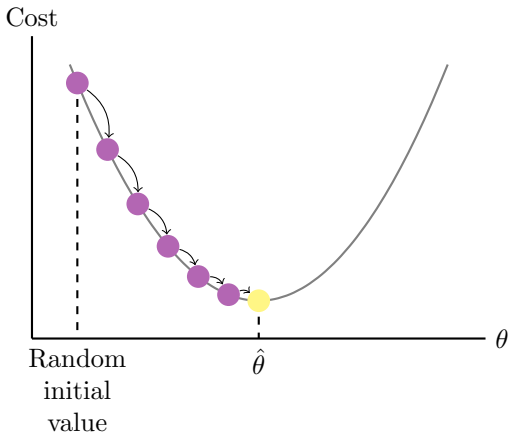


Figure 21: Gradient descent method applied on a simple function.

3.3 Advanced Topics in Neural Networks

3.3.1 Deep Neural Networks

A *deep neural network* (DNN) [Figure 22] is an artificial neural network characterized by multiple hidden layers between the input and output layers. The increased depth allows DNNs to model highly complex patterns and relationships in the data. Each layer in a DNN can be seen as learning a different level of abstraction, with the early layers capturing low-level features such as edges and textures in image data, while deeper layers capture high-level features like shapes and objects. This hierarchical learning capability 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 techniques such as dropout and batch normalization to improve performance and prevent overfitting.

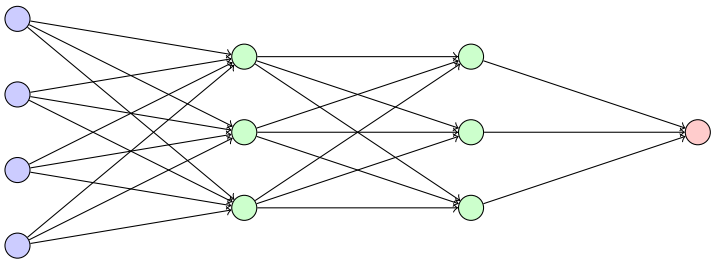


Figure 22: A Deep neural network with two hidden layers.

3.3.2 Convolutional Neural Networks

A *convolutional neural network* (CNN) [Figure 23] is a specialized type of neural network designed for processing structured grid data, like images. CNNs use convolutional layers that apply a series of learnable filters to the input data to extract features. These filters slide over the input data, performing element-wise multiplications and summing the results, which helps in detecting patterns such as edges, textures, and shapes. 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 reduce the spatial dimensions of the data, and fully connected layers, which integrate the extracted features to make the final predictions. CNNs have revolutionized computer vision tasks, achieving state-of-the-art results in image classification, object detection, and segmentation. Advanced architectures like ResNet, Inception, and EfficientNet further enhance the capabilities of CNNs by introducing innovative design elements and optimization techniques.

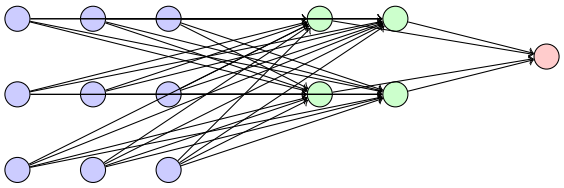


Figure 23: A simple convolutional neural network architecture.

3.3.3 Recurrent Neural Networks

A *recurrent neural network* (RNN) [Figure 24] is a type of neural network that is particularly well-suited for sequential data, such as time series or natural lan-

guage. RNNs have connections that form directed cycles, allowing information to persist. This cyclic structure enables RNNs to maintain a hidden state that captures information from previous time steps, making them effective for tasks where the context is important, such as language modeling, machine translation, and speech recognition. 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. Variants of RNNs, such as Long Short-Term Memory (LSTM) and Gated Recurrent Unit (GRU) networks, address some of the limitations of basic RNNs by introducing mechanisms to learn long-term dependencies and mitigate issues related to vanishing gradients.

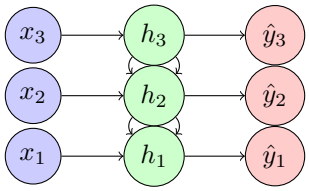


Figure 24: Recurrent Neural Network (RNN) unrolled through time.

3.4 Graph Neural Networks and Attention Mechanisms

3.4.1 Graph Neural Networks

A *graph neural network* (GNN) is an advanced type of neural network designed to handle graph-structured data, where relationships between data points are represented as edges between nodes. GNNs generalize neural networks to work directly on the graph domain by incorporating the graph's structure into the learning process. Nodes in a GNN aggregate feature information from their neighbors through multiple layers, enabling the network to capture the dependencies and relationships inherent in the graph. This message-passing mechanism [Figure 25] allows GNNs to learn rich node representations and 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. Advanced GNN architectures, such as Graph Convolutional

Networks (GCNs), Graph Attention Networks (GATs), and GraphSAGE, leverage different strategies for aggregating and updating node information, further enhancing their capabilities.

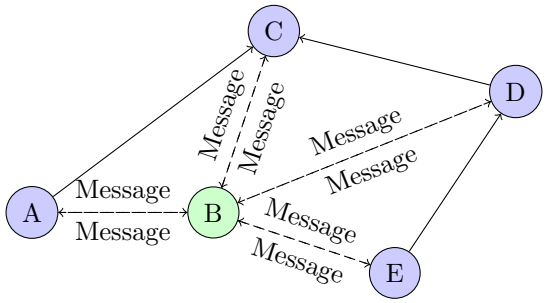


Figure 25: Message passing between node B and its neighbors in a GNN.

3.4.2 Attention Mechanisms

Attention mechanisms are sophisticated techniques used to enhance the performance of neural networks by dynamically focusing on the most relevant parts of the input data while processing. The most prominent use of attention mechanisms is in the *Transformer* model, which has revolutionized natural language processing tasks. The attention mechanism [Figure 26] computes a weighted sum of values, where the weights are derived from the compatibility of the query with the corresponding keys, allowing the model to selectively focus on important parts of the input. This selective focus helps improve the efficiency and accuracy of the model, especially 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. The combination of GNNs and attention mechanisms has shown significant promise in various applications, enabling models to effectively capture complex dependencies and relationships in the data. In particular, the integration of attention mechanisms into GNNs has led to the development of advanced models like the Graph Attention Network (GAT), which assigns different importance to different neighbors during the message-passing process, thereby improving the model's ability to learn from graph-structured data.

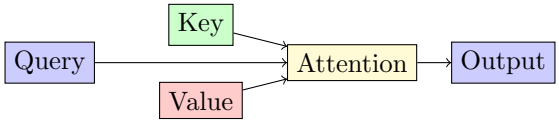


Figure 26: Attention mechanism in neural networks.

4 Graph Similarity Problem

The *graph similarity problem* entails determining the degree of similarity or dissimilarity between two graphs. This problem has vast and varied applications across numerous domains, including pattern recognition, computer vision, bioinformatics, social network analysis, and chemical informatics. In these fields, comparing the structural properties of graphs can yield insights into patterns, relationships, and functional similarities between complex systems. 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. Due to its broad relevance, numerous methods have been developed to quantify graph similarity, each with its own strengths and limitations. Additionally, it is often desirable to retrieve the edit path from one graph to another, alongside the similarity metric, to understand the specific transformations involved. However, we will focus solely on the similarity metrics and will not address the retrieval of edit paths.

In the following sections, we will explore several key methods for measuring 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.

4.1 Graph Isomorphism

Graph isomorphism is one of the fundamental concepts in graph theory 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:

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

While graph isomorphism provides a binary determination of whether two graphs are identical in structure, it is limited because it does not quantify the degree of similarity for non-isomorphic graphs. It is useful in applications where exact structural equivalence is necessary, such as in database indexing, chemical compound comparison, and verifying the correctness of network models. However, it is less useful in cases where graphs are similar but not identical, as it cannot measure partial similarity or small structural differences.

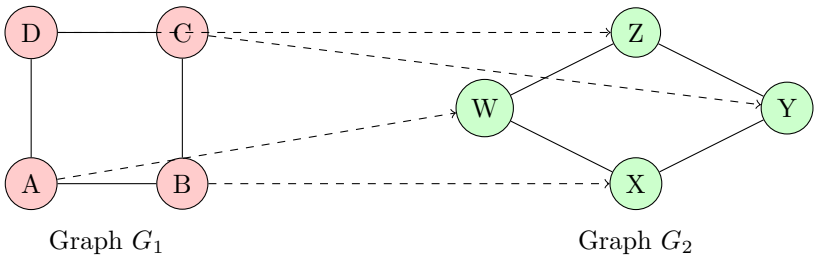


Figure 27: Graph Isomorphism: G_1 (Square) and G_2 (Rhombus).

4.2 Graph Kernels

Another sophisticated approach to measuring graph similarity involves the use of *graph kernels*. Graph kernels provide a way to compute the similarity between two graphs based on their structural attributes and properties. They transform graphs into high-dimensional vectors and then compare these vectors using kernel functions. Common types of graph kernels include:

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

Graph kernels are powerful because they can incorporate various types of structural information and are well-suited for use in machine learning algorithms. They are particularly useful in applications like chemical compound classification, bioinformatics, and social network analysis. However, they can be computationally intensive and require careful tuning of parameters.

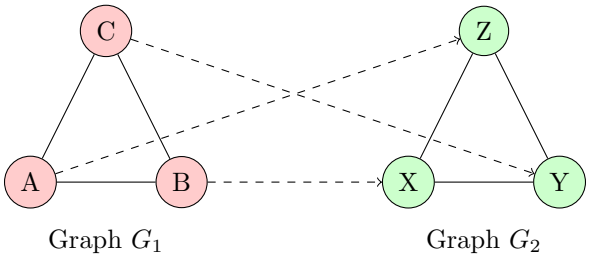


Figure 28: Graph Kernels: Example Graphs with Similar Structures.

4.3 Graph Edit Distance (GED)

The *graph edit distance* (GED) is a more flexible and informative metric for measuring the similarity between two graphs, $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$. It quantifies similarity by determining the minimum cost required to transform G_1 into G_2 using a series of atomic operations. These operations include vertex and edge insertions, deletions, and substitutions. The cost of each operation is determined by a predefined cost function.

Formally, let Σ be the set of all possible edit operations, and let $c : \Sigma \to \mathbb{R}^+$ be a cost function that assigns a positive real number to each operation. The GED 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.

The computation of GED is known to be **NP-HARD**, which means 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 nuanced measure of similarity even when the graphs are not identical. This makes it particularly useful in applications where exact matching is impractical or unnecessary, such as in approximate pattern matching in images or molecules.

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 $e' = \{u', v'\}$.

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

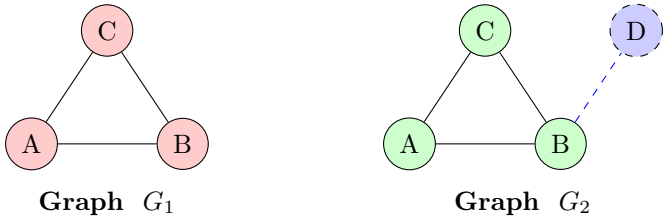


Figure 29: Graph Edit Distance: Transforming G_1 to G_2 by adding vertex D and edge (B,D).

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: $1 + 1 = 2$.

The GED's flexibility and detailed transformation cost make it a powerful tool for measuring graph similarity. While computationally intensive, it provides a meaningful and precise measure of similarity that can capture both small and large differences between graphs, making it highly valuable for applications where nuanced similarity measures are crucial.

5 State of the Art Review

Traditional imperative solutions for solving the graph edit distance (GED) problem have been effective for graphs of modest size, utilizing explicit algorithms that compare graphs node by node and edge by edge through combinatorial techniques. However, as the number of nodes in the graph increases, the complexity of these methods grows exponentially, leading to significant scalability issues. In such scenarios, imperative solutions often become impractical, requiring an inordinate amount of time to compute the GED. This challenge is particularly critical in domains like bioinformatics, social network analysis, and cheminformatics, where large-scale graphs are common.

To address these scalability challenges, recent research has pivoted towards leveraging artificial intelligence (AI) models. AI-based approaches offer more robust and scalable solutions by learning patterns and features from graph data, significantly reducing computation times and improving accuracy. This section reviews some of the prominent AI-based models that have advanced the state of the art in GED computation.

The timeline of notable advancements in this field starts with *SimGNN* in 2019 [2], the first approach to utilize neural networks for computing GED. Subsequent models have built upon this foundation, progressively enhancing the accuracy and efficiency of GED computation. This review covers developments from SimGNN to the latest contributions in 2023, culminating in the VLDB pa-

per introducing *GedGNN* [10]. These models collectively represent significant strides in utilizing AI to overcome the limitations of traditional GED computation methods.

5.1 Timeline

2019, *SimGNN: A Neural Network Approach to Fast Graph Similarity Computation* [2]: Introduces SimGNN, addressing graph similarity computation using neural networks. It features a learnable embedding function, an attention mechanism to emphasize 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* [12]: 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* [15]: 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* [16]: 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* [17]: 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* [14]: 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* [6]: 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* [18]: 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* [1]: 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* [5]: 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* [13]: 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* [9]: 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* [4]: 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* [11]: 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* [8]: 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* [7]: 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 [3]: 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* [10]: 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 significantly outperformed the competition is SimGNN [2], introduced in 2019. SimGNN serves as a foundational model in the field of graph similarity computation, and subsequent models often inherit its core concepts, making it the starting point of reference for anyone working in this niche field.

SimGNN is an end-to-end neural network-based approach designed to learn a function that maps a pair of graphs to a similarity score. An overview of SimGNN is illustrated in Figure 30. The architecture of SimGNN involves several key stages:

- **Node Embedding Stage**: Each node in the graph is transformed into a vector that encodes its features and structural properties. This stage leverages a graph convolutional network to capture local structural information.
- **Graph-Level Embedding Stage**: The node embeddings are aggregated using an attention mechanism to produce a single embedding for each graph. The attention mechanism helps to emphasize more important nodes, improving the overall representation of the graph.
- **Graph-Graph Interaction Stage**: The graph-level embeddings of the two graphs are interacted to produce interaction scores that represent the similarity between the graphs. This interaction is performed through a neural network that learns the relationship between the embeddings.
- **Final Similarity Score Computation Stage**: The interaction scores are further processed to compute the final similarity score, which is compared against the ground-truth similarity score for parameter updates.

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

- **Pairwise Node Comparison**: This strategy involves computing pairwise interaction scores between the node embeddings of the two graphs. For graphs of different sizes, fake nodes with zero embeddings are added

to the smaller graph to ensure compatibility. The resulting similarity matrix is used to extract histogram features, which are then combined with graph-level interaction scores to provide a comprehensive view of graph similarity.

The combination of these two strategies allows SimGNN to capture both coarse global comparison information and fine-grained node-level comparison information, resulting in a robust and thorough approach to graph similarity computation.

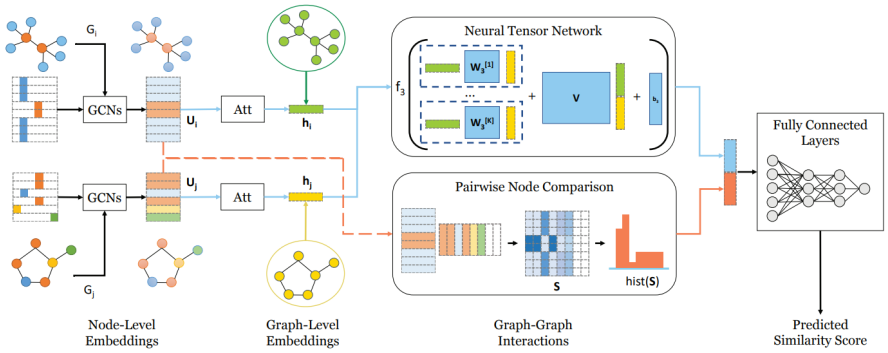


Figure 30: SimGNN architecture overview.

5.3 GPN

The Graph Path Networks (GPN) model, proposed in 2022 within the *NOAH Framework* [17], introduces a novel approach to computing approximate Graph Edit Distance (GED) by leveraging the A* search algorithm optimized through neural networks. This method addresses several limitations found in previous models, including SimGNN, by improving both the search direction and search space optimization.

An overview of GPN is illustrated in Figure 31. The architecture of GPN comprises several key components:

- **Pre-training Module**: This module computes pre-training information such as exact GEDs and graph edit paths. It generates (sub)graph pairs used in training, enriching the model's understanding of graph transformations.
- **Graph Embedding Module**: Utilizing the Graph Isomorphism Network (GIN), this module transforms each node into a vector encoding its features and structural properties. It incorporates cross-graph information and employs an attention mechanism to obtain the final graph-level embeddings.

- **Learning Module**: This component focuses on optimizing the A* search algorithm by learning an estimated cost function and an elastic beam size. The estimated cost function helps guide the search direction, while the beam size adapts to various user settings, improving the search space optimization.

GPN's enhancements over SimGNN include improved search efficiency and accuracy in GED computation. It also facilitates tasks such as graph similarity search and graph classification, demonstrating its versatility and robustness. Notably, GPN is capable of finding an edit path between graphs, utilizing a combination of pre-training information and adaptive search strategies to achieve high performance across different datasets.

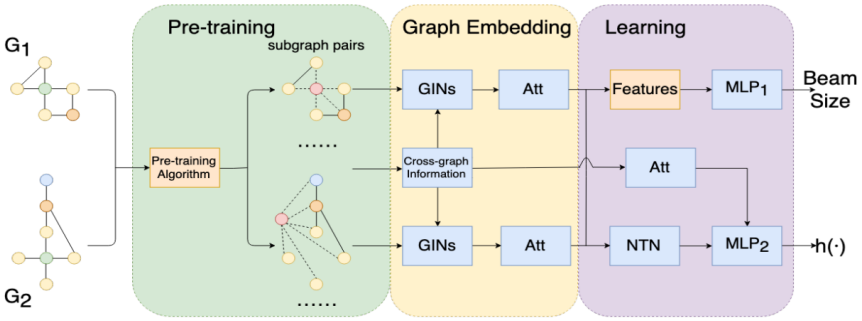


Figure 31: GPN architecture overview.

5.4 TaGSim

TaGSim (Type-aware Graph Similarity) [1], introduced in 2022, represents a significant advancement in the computation of Graph Edit Distance (GED) by addressing the limitations of previous approaches, including those that treat GED as a single global measure. TaGSim refines GED computation by considering the distinct impacts of various types of graph edit operations.

An overview of TaGSim is illustrated in Figure 32. The TaGSim framework operates through the following key components:

- **Type-Aware Graph Embeddings**: TaGSim models the transformative impacts of four specific graph edit operations—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.
- **Graph Edit Value (GEV) Dimensions**: GED is decoupled into four dimensions corresponding to the different types of graph edits. TaGSim estimates each dimension individually, providing a more granular and accurate representation of graph similarity.

- **Type-Aware Neural Networks**: The framework uses 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.

TaGSim's approach offers several advantages over traditional GED computation methods and previous AI-based models. By decoupling the GED into different dimensions, it avoids the pitfalls of treating GED as a single undifferentiated metric. This granularity enhances both the accuracy and interpretability of the similarity scores, making TaGSim a robust solution for various graph analysis tasks.

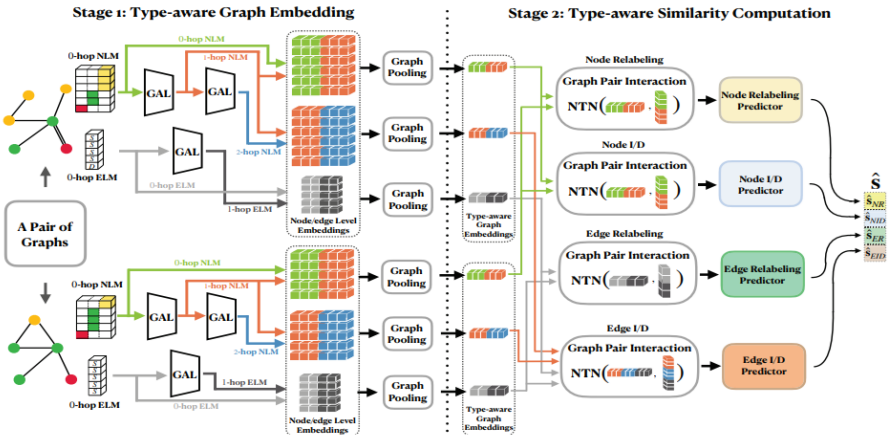


Figure 32: TaGSim architecture overview.

5.5 GedGNN

GedGNN (Graph Edit Distance via Neural Graph Matching) [10], introduced in 2023, represents the latest advancement in the field of Graph Edit Distance (GED) computation. This model addresses the challenge of capturing both node and edge matching effectively through a sophisticated neural network architecture.

An overview of GedGNN is illustrated in Figure 33. The GedGNN architecture comprises several innovative components:

- **Graph Neural Network (GNN) Encoder**: This component uses a GNN to encode the structural and attribute information of the input graphs. The encoder generates embeddings for nodes and edges, preserving their relational information.
- **Node and Edge Matching Module**: This module performs node and edge matching between the two graphs. It uses an iterative matching algorithm to refine the matches, improving accuracy over multiple iterations.

- **k-Best Matching Post-Processing Algorithm**: To further enhance the matching accuracy, GedGNN incorporates a k-best matching algorithm. This algorithm selects the best k matching solutions from the initial matches, refining the final GED computation.

GedGNN's comprehensive approach ensures high accuracy in GED estimation by combining robust embedding techniques with advanced matching algorithms. This model not only outperforms previous methods but also provides a flexible framework that can adapt to various types of graph structures and similarity measures.

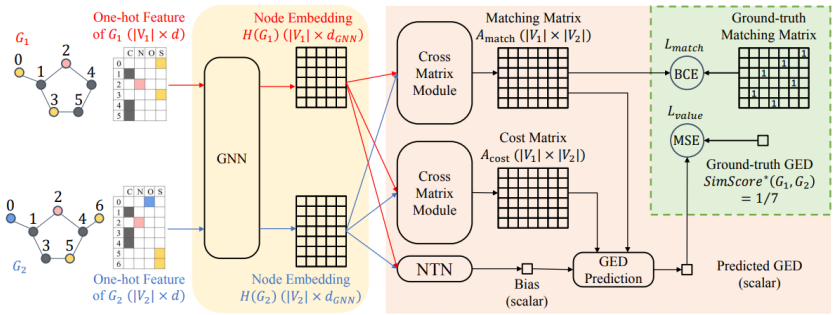


Figure 33: GedGNN architecture overview.

5.6 Encountered Gaps

In working with the code for GedGNN, several critical gaps have become evident, particularly concerning the practical implementation and evaluation of the model. These gaps hinder both the usability of the code and the reliability of the results obtained. A primary issue is the lack of clear and well-defined parameters, which severely impacts the reproducibility of results. The instructions provided are often vague, with key hyperparameters like learning rates, regularization factors, and model configurations either poorly documented or omitted entirely. This makes it challenging to replicate the experiments accurately and to achieve the reported performance in different environments.

The quality of the code itself presents another significant challenge. The codebase is poorly documented and lacks clarity and organization, making it difficult to read, understand, and refactor. Attempting to extend or modify the code proved particularly challenging due to the absence of best practices in software development, such as modular design and comprehensive commenting. This not only stifles innovation but also makes it difficult to ensure that any changes do not introduce new issues or degrade performance.

Furthermore, the scalability of the code is limited. While the code is designed to run on GPUs, it does not scale well, failing to deliver the significant speedups that one might expect. This inefficiency further complicates the practical appli-

cation of GedGNN, particularly for large-scale problems where computational efficiency is crucial.

Another critical concern is the fairness of the model's evaluation. The benchmarks used in evaluating GedGNN may not fully capture the model's true performance, as they often do not represent real-world scenarios or test the model's robustness across diverse graph types and sizes. This raises questions about whether the model is genuinely being tested for its effectiveness or if the evaluations are biased towards specific datasets or problem formulations. Such limitations can lead to inflated performance claims that do not generalize well to other, more varied graph datasets.

Lastly, the quality of the data used for training and evaluating the model is problematic. The datasets employed often contain approximate GED values rather than the true edit distances, introducing inaccuracies that can mislead the training process. Additionally, these datasets are typically small and specialized, focusing on a single type of graph application, such as chemical compounds or social networks. As a result, the model tends to perform poorly when applied to different types of graphs that were not represented in the training data, indicating a lack of generalizability.

In summary, the code for GedGNN presents several significant challenges that hinder its practical utility and broader adoption. These include issues with reproducibility due to unclear parameters, poor code quality that complicates refactoring and extension, limited scalability, unfair model evaluations, and reliance on low-quality, non-generalizable datasets. Addressing these issues is crucial for advancing the field and ensuring that GED models like GedGNN can be reliably and effectively applied in diverse real-world scenarios.

6 Methodology and Experimentation

7 Discussion and Conclusions

References

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

[2] 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.

[3] 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.

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

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

[6] 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.

[7] 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.

[8] 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.

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

[10] 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.

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

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

[13] 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.

[14] 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.

[15] 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.

[16] 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.

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

[18] 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.