GRAIL: GRAPH EDIT DISTANCE AND NODE ALIGNMENT USING LLM-GENERATED CODE

Samidha Verma[1*] **Arushi Goyal**[1*] **Ananya Mathur**[1*] **Ankit Anand**[2] **Sayan Ranu**[1]

[1]Indian Institute of Technology Delhi, India [2]Google DeepMind, Montreal, Canada

{samidha.verma@scai,cs5200418@,cs5200416@,sayanranu@}.iitd.ac.in,
anandank@google.com

ABSTRACT

Graph Edit Distance (GED) is a widely used metric for measuring similarity between two graphs. Computing the optimal GED is NP-hard, leading to the development of various neural and non-neural heuristics. While neural methods have achieved improved approximation quality compared to non-neural approaches, they face significant challenges: *(1)* They require large amounts of ground truth data, which is itself NP-hard to compute. *(2)* They operate as black boxes, offering limited interpretability. *(3)* They lack cross-domain generalization, necessitating expensive retraining for each new dataset. We address these limitations with GRAIL, introducing a paradigm shift in this domain. Instead of training a neural model to predict GED, GRAIL employs a novel combination of large language models (LLMs) and automated prompt tuning to generate a *program* that is used to compute GED. This shift from predicting GED to generating programs imparts various advantages, including end-to-end interpretability and an autonomous self-evolutionary learning mechanism without ground-truth supervision. Extensive experiments on seven datasets confirm that GRAIL not only surpasses state-of-the-art GED approximation methods in prediction quality but also achieves robust cross-domain generalization across diverse graph distributions.

1 INTRODUCTION AND RELATED WORK

Graph Edit Distance (GED) quantifies the dissimilarity between two graphs as the minimum number of *edits* required to transform one graph into another. An edit may comprise adding or deleting nodes and edges or replacing node and edge labels. Fig. 1 presents an example. Computing GED is NP-hard Ranjan et al. (2022) and APX-hard Fan et al. (2020). Owing to its numerous applications Blumenthal (2019); Ranu et al. (2014); Bommakanti et al. (2024b), polynomial-time heuristics are designed in practice.

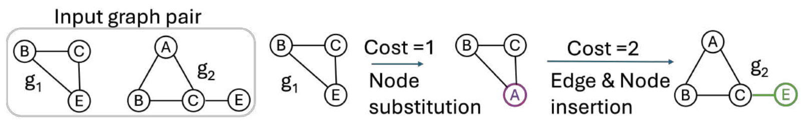


Figure 1: Illustration of edit path from g_1 to g_2 with GED 3.

1.1 EXISTING WORKS AND THEIR LIMITATIONS

Existing heuristics to approximate GED can be broadly grouped into two paradigms: non-neural and neural.

Non-Neural Methods: A comprehensive survey on non-neural methods is available at Blumenthal et al. (2020). These approaches leverage techniques such as transformations to the linear sum assignment (NODE Justice & Hero (2006), BRANCH-TIGHT Blumenthal & Gamper (2018)), mixed integer programming (MIP) (LP-GED-F2 Lerouge et al. (2017a), ADJ-IP Justice & Hero (2006), COMPACT-MIP Blumenthal & Gamper (2020)), local search methods (IPFP Leordeanu et al. (2009)), and approximations to the quadratic asssignment problem Bommakanti et al. (2024a).

*Denotes equal contribution.

Name	End-to-end interpretable	Cross-domain generalization	Non-reliant on NP-hard supervision	Accurate
GREED Ranjan et al. (2022)	✗	✗	✗	✓
GEDGNN Piao et al. (2023)	○	✗	✗	✓
H2MN Zhang et al. (2021)	✗	✗	✗	✓
ERIC Zhuo & Tan (2022)	✗	✗	✗	✓
GRAPHEDX Jain et al. (2024)	✗	✗	✗	✓
GRAPHOTSIM Doan et al. (2021)	✗	✗	✗	✓
GRAPHSIM Bai et al. (2020)	✗	✗	✗	✓
TAGSIM Bai & Zhao (2021)	✗	✗	✗	✓
GMN Li et al. (2019)	✗	✗	✗	✓
GENN-A* Wang et al. (2021)	○	✗	✗	✓
SIMGNN Bai et al. (2019)	✗	✗	✗	✓
Non-neural approaches Blumenthal et al. (2020)	○	✓	✓	✗
GRAIL	✓	✓	✓	✓

Table 1: Summary of the drawbacks of existing algorithms and the proposed algorithm GRAIL. ✓ indicates satisfaction of a desirable property, ✗ indicates non-satisfaction, and ○ indicates partial satisfaction. GEDGNN, GENN-A*, and traditional non-neural approaches achieve partial interpretability by providing edit paths corresponding to the GED, but they do not explain the semantic reasoning behind these paths. In contrast, GRAIL achieves end-to-end interpretability through its code-based output, where each decision can be traced to its underlying logical reasoning. Non-neural approaches utilize unsupervised learning, enabling cross-domain generalization. However, their approximation errors are significantly higher on average than neural approaches, as demonstrated in § 5.

Neural Methods: Recent literature shows a shift towards graph neural network-based approaches for approximating GED, driven by their superior approximation quality compared to non-neural methods Ranjan et al. (2022); Zhang et al. (2021); Bai et al. (2019); Piao et al. (2023); Wang et al. (2021); Zhuo & Tan (2022); Jain et al. (2024); Bai et al. (2020); Doan et al. (2021); Li et al. (2019). However, these advancements come with limitations, as summarized in Table 1.

- **Lack of interpretability:** Most neural methods only predict the GED and not the corresponding edit path. The edit path is essential for various applications such as identifying functions of protein complexes Singh et al. (2008), image alignment Conte et al. (2003), and uncovering gene-drug regulatory pathways Chen et al. (2019). Few neural methods that predict the edit path Piao et al. (2023); Wang et al. (2021) rely on expensive ground truth computation, which can only be attained for very small graphs ($\approx$ 10 nodes). For larger graphs, random edits are made to synthetic graphs to generate the training samples.
- **NP-hard training data:** The training dataset for neural methods consists of graph pairs and their true GED. GED computation is NP-hard. Therefore, generating this training data is prohibitively expensive and restricted to small graphs only. Hence, approximation error deteriorates on larger graphs. Ranjan et al. (2022)
- **Lack of generalization:** Neural GED approximators struggle to generalize across datasets. For datasets from different domains (e.g., chemical compounds vs. function-call graphs), the node label sets often differ. Since the number of parameters in a GNN depends on the feature dimensions of the nodes, GNNs fail to generalize across domains. Even within the same domain, as demonstrated later in §5, distribution shifts in structural and node label distributions lead to increased approximation error. This limitation necessitates generating ground-truth data and training separate models for each dataset. Given that generating training data is NP-hard, this pipeline becomes highly resource-intensive.

1.2 CONTRIBUTIONS

We address the above-outlined limitations through GRAIL: Graph Edit Distance and Node Alignment using LLM-Generated Code. GRAIL introduces a paradigm shift in the domain of GED approximations through the following novel innovations.

- **Problem formulation:** We shift the learning objective from approximating GED to learning a *program* that approximates GED. This reformulation provides end-to-end interpretability, as each algorithmic decision can be traced to its underlying logical reasoning. Moreover, by elevating the output to a higher level of abstraction through code generation, we achieve superior generalization across datasets, domains, graph sizes, and label distributions.
- **LLM-guided program discovery:** The algorithmic framework of GRAIL is grounded on three novel design choices. First, we map the problem of approximating GED to *maximum weight bipartite matching*, where the weights of the bipartite graph are computed using an LLM-generated program. Second, the prompt provided to the LLM is tuned through an evolutionary algorithm Romera-Paredes et al. (2024). Third, our prompt-tuning methodology eliminates the need for ground-truth GED data by designing a prediction framework where the prediction is guaranteed to be an *upper*

bound to the true GED. Hence, minimizing the upper-bound is equivalent to minimizing the approximation error, thereby overcoming a critical bottleneck of existing neural approaches.

- **Comprehensive Empirical Evaluation:** Through extensive experiments across 6 datasets, we demonstrate that GRAIL discovers *foundational* code-based heuristics. Specifically, these heuristics not only surpass the state-of-the-art methods in GED computation but also exhibit generalization across diverse datasets and domains. This crucial feature eliminates the need for costly dataset-specific training, thereby addressing a significant limitation of existing neural algorithms.

2 PRELIMINARIES AND PROBLEM FORMULATION

Definition 1 (Graph). *We represent a node-labeled undirected graph as $\mathcal{G}(\mathcal{V}, \mathcal{E}, \mathcal{L})$ where $\mathcal{V} = \{v_1, \cdots, v_{|\mathcal{V}|}\}$ is the set of nodes, $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$ is the edge set and $\mathcal{L} : \mathcal{V} \rightarrow \Sigma$ is a labeling function that maps nodes to labels, where Σ is the set of all labels.*

In unlabeled graphs, all nodes are assigned the same label.

Definition 2 (Node Mapping). *A node mapping between two graphs $\mathcal{G}_1$ and $\mathcal{G}_2$, each consisting of n nodes, refers to a bijection $\pi : \mathcal{V}_1 \rightarrow \mathcal{V}_2$, where every node $v \in \mathcal{V}_1$ is uniquely mapped to a node $\pi(v) \in \mathcal{V}_2$.*

Extension to graphs of different sizes: When dealing with two graphs $\mathcal{G}_1$ and $\mathcal{G}_2$ with different numbers of nodes, n_1 and n_2 respectively, such that $n_1 < n_2$, the smaller graph $\mathcal{G}_1$ can be extended to match the size of $\mathcal{G}_2$ by introducing $n_2 - n_1$ additional isolated *dummy* nodes. These new nodes are labeled with a unique identifier, ϵ, indicating that they are placeholders with no connections. From this point onward, we assume that any pair of graphs in consideration have an equal number of nodes, with smaller graphs being augmented by dummy nodes as necessary.

Definition 3 (GED under a node mapping π). *Given a node mapping π, the cost function for calculating graph edit distance between graphs $\mathcal{G}_1(\mathcal{V}_1, \mathcal{E}_1, \mathcal{L}_1)$ and $\mathcal{G}_2(\mathcal{V}_2, \mathcal{E}_2, \mathcal{L}_2)$ is expressed as:*

$$\begin{aligned}\mathrm{GED}_\pi(\mathcal{G}_1, \mathcal{G}_2) = &\sum_{v_1 \in \mathcal{V}_1} \mathbb{I}\left(\mathcal{L}_1(v_1) \neq \mathcal{L}_2(\pi(v_1))\right) \\ + \frac{1}{2} \sum_{v_1 \in \mathcal{V}_1} &\sum_{v_2 \in \mathcal{V}_1} \mathbb{I}\left(e_1(v_1, v_2) \neq e_2(\pi(v_1), \pi(v_2))\right)\end{aligned}$$

where,

- *$e_i(u, v)$ returns 1 if the edge $(u, v) \in \mathcal{E}_i$ in graph $\mathcal{G}_i$, 0 otherwise.*
- *$\mathbb{I}(A)$ is the indicator function, which is 1 if the condition A holds, and 0 otherwise.*

Interpretation of edit path from node mapping: The first part of the equation captures node mismatches where it evaluates the label differences between nodes in $\mathcal{G}_1$ and $\mathcal{G}_2$. Mapping a dummy node to a real node (or vice versa) results in a label mismatch, reflecting the insertion or deletion of a node, while a mismatch between real nodes denotes a substitution. The second part of the equation captures edge mismatches. Specifically, if an existing edge in $\mathcal{G}_1$ (i.e., $e_1(v_1, v_2) = 1$) is mapped to a non-existing edge in $\mathcal{G}_2$ (i.e., $e_2(\pi(v_1), \pi(v_2)) = 0$) or vice versa, the cost is 1 representing edge deletion and insertion, respectively.

Definition 4 (Graph edit distance (GED)). *The GED between graphs $\mathcal{G}_1$ and $\mathcal{G}_2$ is the minimum GED across all possible node mappings.*

$$\mathrm{GED}(\mathcal{G}_1, \mathcal{G}_2) = \min_{\forall \pi \in \mathcal{M}} \{\mathrm{GED}_\pi(\mathcal{G}_1, \mathcal{G}_2)\} \tag{1}$$

Here, $\mathcal{M}$ denotes the universe of all possible mappings.

The problem is hard (NP-hard and APX-hard) since the cardinality of $\mathcal{M}$ is $n!$, where $n = \max\{|\mathcal{V}_1|, |\mathcal{V}_2|\}$.

The problem of learning to code for approximating GED is defined as follows.

Problem 1 (Learning to code for GED). *Given a set of training graph pairs $\mathbb{T} = \{\langle \mathcal{G}_1, \mathcal{G}'_1 \rangle, \langle \mathcal{G}_2, \mathcal{G}'_2 \rangle, \cdots, \langle \mathcal{G}_n, \mathcal{G}'_n \rangle\}$, learn a program $P : (\mathcal{G}_t, \mathcal{G}'_t) \rightarrow \mathbb{Z}+$ that takes as input a graph pair $\langle \mathcal{G}_t, \mathcal{G}'_t \rangle \in \mathbb{T}$, and outputs a non-negative integral distance that minimizes*

$$\sum_{t=1}^{n} |P(\mathcal{G}_t, \mathcal{G}'_t) - \mathrm{GED}(\mathcal{G}_t, \mathcal{G}'_t)| \tag{2}$$

Note that our training set consists solely of graph pairs, without requiring their true GED, which is computationally prohibitive due to its NP-hardness. As we will elaborate in the next section, we identify polynomial-time computable upper bounds for the true GED and reformulate the optimization objective to minimize this upper bound. This autonomous self-evolutionary learning mechanism overcomes a significant limitation of neural GED approximators.

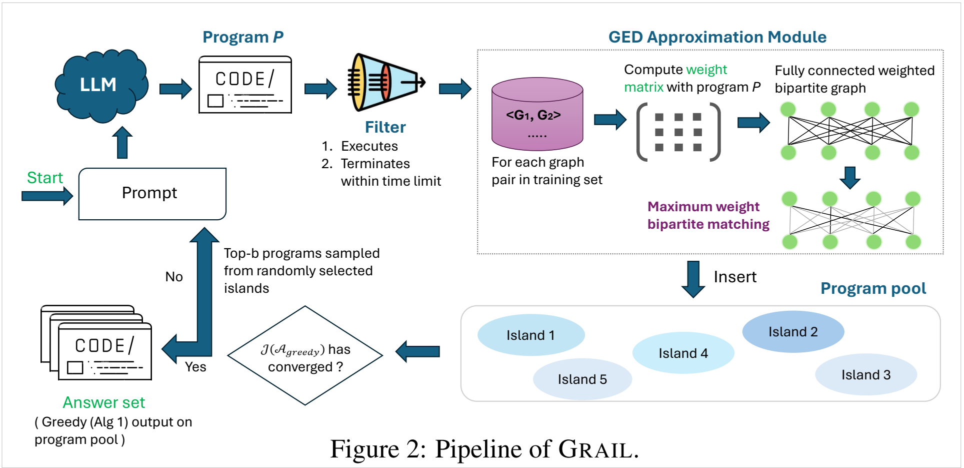

Program P
LLM
CODE/
Filter
1. Executes
2. Terminates within time limit
GED Approximation Module
<G1, G2>
For each graph pair in training set
Compute weight matrix with program P
Fully connected weighted bipartite graph
Maximum weight bipartite matching
Start
Prompt
Top-b programs sampled from randomly selected islands
No
Yes
Insert
Program pool
Island 1
Island 2
Island 3
Island 4
Island 5
$\mathcal{J}(\mathcal{A}_{greedy})$ has converged ?
Answer set
(Greedy (Alg 1) output on program pool)

Figure 2: Pipeline of GRAIL.

3 APPROXIMATION STRATEGY

The true GED corresponds to the minimum distance across all possible node mappings (Def. 4). However, enumerating all such mappings is computationally infeasible due to its factorial complexity relative to graph size. To overcome this challenge, we approximate the GED by evaluating a small subset of mappings (e.g., 15) and selecting the minimum distance among them. These mappings are generated by programs derived from the LLM, as detailed in § 4. Importantly, this approximated GED serves as an upper bound to the true GED, as it considers only a subset of all possible node mappings.

3.1 NODE MAPPINGS THROUGH BIPARTITE MATCHING

The task of mapping nodes between two graphs can be approximated as *Maximum Weight Bipartite Matching*.

Definition 5 (Maximum Weight Bipartite Matching)**.** *Given a weighted bipartite graph $\mathcal{B}(\mathcal{V},\mathcal{U},\mathcal{E},\mathcal{W})$ with node sets $\mathcal{V}$ and $\mathcal{U}$, and a weighted edge set $\mathcal{E}:\mathcal{V}\times\mathcal{U}\rightarrow\mathbb{R}$ where $\mathcal{W}:\mathcal{E}\rightarrow\mathbb{R}$ assigns weights to edges, find a subset of edges $\mathcal{E}^*\subseteq\mathcal{E}$ that **(1)** induces a bijection between nodes in $\mathcal{V}$ and nodes in $\mathcal{U}$, and **(2)** maximizes the total weight of the mapped edges, i.e., $\sum_{e\in\mathcal{E}^*}\mathcal{W}(e)$. Here, $\mathcal{W}(e)$ represents the weight of edge e.*

Maximum weight bipartite matching can be solved optimally in polynomial time using the Hungarian algorithm Kuhn (1955). Additionally, several heuristics have been proposed, such as the Hopcroft–Karp Algorithm Hopcroft & Karp (1973), the Neighbor-biased mapper He & Singh (2006), or greedy selection of the highest-weight edges while maintaining the one-to-one mapping constraint. We use the notation $\pi(\mathcal{B})$ to denote the node mapping obtained from $\mathcal{B}$.

To use maximum weight bipartite matching for approximating GED, for given graphs $\mathcal{G}_1(\mathcal{V}_1,\mathcal{E}_1,\mathcal{L}_1)$ and $\mathcal{G}_2(\mathcal{V}_2,\mathcal{E}_2,\mathcal{L}_2)$, we construct a fully connected, weighted bipartite graph $\mathcal{B}(\mathcal{V}_1,\mathcal{V}_2,\mathcal{E},\mathcal{W})$ where edge set $\mathcal{E}=\{(v_1,v_2)\mid v_1\in\mathcal{V}_1,v_2\in\mathcal{V}_2\}$[1]. The weight of an edge (v_1,v_2) is set based on some policy, which should ideally reflect the probability of v_1 being mapped to v_2 in the optimal GED mapping. Maximum weight bipartite matching is then performed on $\mathcal{B}$ using any standard algorithm, and the GED is computed based on the mapping $\pi(\mathcal{B})$.

The quality of the mapping with respect to approximating GED, therefore, rests on the edge weights in the bipartite graph. We will use an LLM to learn the policy, in the form of a program, with the following *minimization objective*.

Problem 2 (Weight Matrix Generation)**.** *Given train set $\mathbb{T}=\{\langle\mathcal{G}_1,\mathcal{G}'_1\rangle,\cdots,\langle\mathcal{G}_n,\mathcal{G}'_n\rangle\}$, generate a program P that takes as input each pair $\langle\mathcal{G}_t,\mathcal{G}'_t\rangle\in\mathbb{T}$ and outputs a corresponding weight matrix $\mathbf{W}_t\in\mathbb{R}^{|\mathcal{V}_t|\times|\mathcal{V}'_t|}$ minimizing*

$$\sum_{t=1}^{n}\text{GED}_{\pi(P)}(\mathcal{G}_t,\mathcal{G}'_t) \tag{3}$$

Here, $\mathbf{W}_t[i,j]$ denotes the weight of edge (v_i,v_j) where $v_i\in\mathcal{V}_t$ to node $v_j\in\mathcal{V}'_t$. $\pi(P)$ denotes the mapping generated by maximum weight bipartite matching on the bipartite graph formed by P.

[1]Recall, we assume the smaller graph is padded with dummy nodes to ensure $|\mathcal{V}_1|=|\mathcal{V}_2|$.

3.2 Budget-constrained Selection of Node Maps

Let $\mathcal{D} = \{P_1, \cdots, P_m\}$ be the set of programs generated by the LLM and $\pi(P_i)$ denote the mapping produced by program $P_i \in \mathcal{D}$. From Def. 4, we know $\text{GED}_{\pi(P_i)}(\mathcal{G}_1, \mathcal{G}_2) \geq \text{GED}(\mathcal{G}_1, \mathcal{G}_2)$, i.e., each program provides an *upper bound* on the true GED. The smaller the upper bound, the closer we are to the true GED. Our goal is to select a subset $\mathcal{A}^* \subseteq \mathcal{D}$ of b programs that minimize the cumulative upper bounds across all train graph pairs. $b \ll |\mathcal{D}|$ denotes the maximum number of mappings we are allowed to evaluate. These b programs will finally be used during inference for unseen graph pairs. Formally, this presents us with the following optimization problem.

Problem 3 (Map Selection)**.** *Given programs $\mathcal{D} = \{P_1, \cdots, P_m\}$, select $\mathcal{A}^*$ such that:*

$$\mathcal{A}^* = \arg \min_{\forall \mathcal{A} \subseteq \mathcal{D},\, |\mathcal{A}| = b} \{\mathcal{J}(A)\} \tag{4}$$

$$\mathcal{J}(\mathcal{A}) = \sum_{\langle \mathcal{G}_1, \mathcal{G}_2 \rangle \in \mathbb{T}} \min_{P \in \mathcal{A}} \left\{ \text{GED}_{\pi(P)}(\mathcal{G}_1, \mathcal{G}_2) \right\} \tag{5}$$

$\mathcal{J}(\mathcal{A})$ quantifies the quality of the subset of mappings in $\mathcal{A}$.

Theorem 3.1. *Prob. 3 is NP-hard.*

Proof. The proposed optimization problem reduces to the *Set Cover* problem Cormen et al. (2009), rendering it NP-hard. For the formal proof, please refer to App. A.1.1. ☐

Owing to NP-hardness, finding the optimal subset of mappings $\mathcal{A}^*$ is not feasible in polynomial time. We establish that $\mathcal{J}(\mathcal{A})$ is *monotonic* and *submodular* (refer App. A.1.2). This enables us to use the greedy hill-climbing algorithm (Alg. 1) to select a sub-optimal but reasonable subset of programs, $\mathcal{A}_{greedy}$. Alg. 1 iteratively selects the program inducing the highest *marginal* reduction in $\mathcal{J}(\mathcal{A}_{greedy})$ till budget is exhausted.

4 Grail: Proposed Methodology

In § 3, we decompose Prob. 1 into two subproblems: weight selection in a bipartite graph (Prob. 2) and budget-constrained map selection (Prob. 3). Prob. 3 is solved (approximately) using Alg. 1. Hence, to complete our approximation scheme, we need to solve Prob. 2.

Fig. 2 presents the pipeline of GRAIL. The process begins with an initial prompt that specifies a trivial program for weight selection, and the LLM is tasked with improving this program for GED computation via bipartite matching (details in § 4.1). Each newly generated program is verified for syntactic correctness and must terminate within a predefined time limit. If these criteria are met, the program is evaluated on the training set of graph pairs and added to the program pool along with its *score*, which reflects its marginal contribution to $\mathcal{J}(\mathcal{A}_{greedy})$. A new prompt is then constructed by sampling the highest-scoring programs from the current pool. The LLM refines these programs, generating new candidates to further enhance performance. These newly generated programs are evaluated and added to the pool following the same procedure. This iterative process continues until $\mathcal{J}(\mathcal{A}_{greedy})$ converges, ensuring that improvements stabilize across iterations. The following sections detail each step of this process.

4.1 Prompt Specification

The prompt is a computer program consisting of three distinct components: **(1)** the problem description, **(2)** the task specification, and **(3)** the top-k programs generated so far based on a scoring function. A sample prompt is provided in Fig. 4 in the Appendix. Here, k is set to 2.

Algorithm 1 The greedy approach

```
Require: Train data $\mathbb{T} = \{T_1, \cdots, T_n\}$ where $T_t = \langle \mathcal{G}_t, \mathcal{G}'_t \rangle$ is a pair of graphs, budget $b$.
Ensure: solution set $\mathcal{A}_{greedy}$, $|\mathcal{A}_{greedy}| = b$
1: $\mathcal{A}_{greedy} \leftarrow \emptyset$
2: while $size(\mathcal{A}_{greedy}) \leq b$ (within budget) do
3:     $P^* \leftarrow \arg\max_{P \in \mathcal{D} \setminus \mathcal{A}_{greedy}} \{\mathcal{J}(\mathcal{A}_{greedy} \cup \{P\}) - \mathcal{J}(\mathcal{A}_{greedy})\}$
4:     $\mathcal{A}_{greedy} \leftarrow \mathcal{A}_{greedy} \cup \{P^*\}$
5: end while
6: Return $\mathcal{A}_{greedy}$
```

Problem Description: The problem description includes the definition of GED, which is embedded as a comment within the program (refer to Fig. 4).

Task Specification: The LLM's task is defined through a comment specifying the inputs it should expect and the required output. The output is a weight matrix $\mathbf{W} \in \mathbb{R}^{|\mathcal{V}_1| \times |\mathcal{V}_2|}$ for the bipartite graph, where $\mathbf{W}[i,j]$ quantifies the strength of mapping node $v_i \in \mathcal{V}_1$ to node $v_j \in \mathcal{V}_2$ in the context of GED computation. The input includes the graph pair represented by their adjacency matrices and an initial weight matrix $\mathbf{W}^0 \in \mathbb{R}^{|\mathcal{V}_1| \times |\mathcal{V}_2|}$ with the same dimensions and semantics as the output. During execution, the input weight matrix $\mathbf{W}^0$ is initialized such that $\mathbf{W}^0[i,j] = 1$ if the corresponding nodes share the same label, i.e., $\mathcal{L}_1(v_i) = \mathcal{L}_2(v_j)$ for $v_i \in \mathcal{V}_1$ and $v_j \in \mathcal{V}_2$, and $\mathbf{W}^0[i,j] = 0$ otherwise. Additionally, the header of the function that the LLM needs to generate is explicitly provided.

Top-k Programs: The initial prompt includes a trivial program where $\forall v_i \in \mathcal{V}_1, v_j \in \mathcal{V}_2, \mathbf{W}[i,j] = 0$. In subsequent iterations, k high-scoring programs are sampled for inclusion in the prompt, where k is a hyper-parameter. The scoring and sampling methodology are described in § 4.2.

4.2 Prompt Tuning

Filter: After a program is generated, it undergoes a filtering step to verify that it executes and terminates on training graph pairs within a predefined time limit. Programs that fail this filter are discarded. For those that pass, we compute their score and add them to our program database $\mathcal{D}$.

Score computation: In Prob. 3, we take the minimum GED across all selected mappings in the answer set. A program's utility, therefore, depends on how it complements other programs in the answer set. Hence, we define its score as the *marginal contribution* to the objective function $\mathcal{J}(\mathcal{A}_{\text{greedy}})$. Specifically, we execute Alg. 1 on the current pool of programs, where $\mathcal{A} = \{P_1, \ldots, P_i\}$ represents the subset of programs selected up to iteration i. If program P is added in the $i+1$-th iteration due to providing the highest marginal contribution, its score is computed as:

$$\text{score}(P) = \mathcal{J}(\mathcal{A} \cup \{P\}) - \mathcal{J}(\mathcal{A}), \tag{6}$$

Evolutionary program selection: The next stage involves selecting programs from the pool to be included in the next prompt. We use the evolutionary algorithm proposed in Funsearch Romera-Paredes et al. (2024) for evolving our programs generated by LLM. Since the programs evolve through mutations introduced by the LLM, the selection mechanism optimizes two distinct objectives. First, the sampled programs should have high scores. Second, the sampled programs should have smaller length improving the interpretability of generated programs.

The evolutionary algorithm follows the *islands model* Gordon & Whitley (1993). Specifically, the population of existing programs is partitioned into s islands, where s is a hyperparameter. Initially, all islands are empty. When a program is added to the pool, it is randomly assigned to an island. Subsequently, to decide which k programs are included in the prompt, we randomly choose an island. Similar to Funsearch, the programs within each island are then split into clusters depending on score. After selecting the island, clusters are selected based on softmax distribution on score. Within a clusters, the programs are selected based on length (smaller is better). Hence, the program selection mechanism for the next prompt favors higher scores and shorter lengths. More details of the process can be found in Romera-Paredes et al. (2024). The LLM is then tasked with further improving these programs. With this design, each island evolves independently. To enable cross-fertilization among islands, we periodically discard half of the islands which have the lowest score. The discarded islands are replaced by iterating over each of the surviving islands, and selecting its best program to seed the replacement population.

4.3 Training and Inference

Training: As illustrated in Fig. 2, each iteration involves generating a program, scoring it, and assigning it to an island before constructing and executing a new prompt. The quality of the program pool is measured by $\mathcal{J}(\mathcal{A}_{greedy})$, serving as an analog to a loss function in our framework. This iterative process continues until $\mathcal{J}(\mathcal{A}_{greedy})$ converges, defined as its improvement over the last i iterations falling below a predefined threshold, akin to the *patience* parameter in neural model training.

Overall, GRAIL seeks to minimize the upper bound of GED. With this strategy, we bypass the need for ground-truth GED data, a key bottleneck in training neural approaches. This unique design is not feasible in neural pipelines since the prediction can err on either side of the true distance.

Inference: During inference, we directly return $\mathcal{J}(\mathcal{A}_{greedy})$ for the given input graph pair. Note that since the output of the training phase is executable code, inference is CPU-bound, enabling it to operate in low-resource environments.

5 EXPERIMENTS

In this section, we benchmark GRAIL and establish that:

- **Approximation Error:** GRAIL achieves low approximation errors and consistently ranks among the top algorithms across all six datasets. Notably, unlike neural approximators, it achieves this performance without relying on extensive NP-hard ground-truth GED training data.
- **Foundational heuristics:** GRAIL breaks new ground by generating heuristics that generalize across diverse datasets, including those featuring unseen node labels and varying graph sizes. This exceptional adaptability sets GRAIL apart, as no existing neural GED approximators have demonstrated such versatility.

The codebase of GRAIL and the programs generated for the various datasets are available at `https://github.com/idea-iitd/Grail`.

Type	Methods	AIDS	Linux	IMDB	ogbg-molhiv	ogbg-code2	ogbg-molpcba	Avg. Rank
LLM	GRAIL	0.57	0.13	0.55	2.96*	4.22	3.18	2
	GRAIL-MIX	0.64	0.11	0.53	2.96	4.10	3.40	2.17
Neural	GREED	0.61	0.41	4.8	3.02	5.52	2.48	3.5
	GEDGNN	0.92	0.29	4.43	1.75	16.68	4.58	5
	ERIC	1.08	0.30	42.44	3.56	17.55	2.79	6.5
	H^2MN	1.14	0.60	57.8	12.01	11.96	5.50	8.33
	GRAPHEDX	0.78	0.27	32.36	14.14	21.46	10.01	8.33
Non Neural	ADJ-IP	0.85	0.50	42.18	10.21	14.94	8.06	7.33
	NODE	2.71	1.24	61.03	4.97	8.34	4.94	8.17
	LP-GED-F2	1.96	0.23	55.26	12.86	16.03	10.30	8.83
	BRANCH	3.31	2.45	7.36	9.86	12.64	11.31	9.33
	COMPACT-MIP	2.69	0.44	65.88	10.88	19.46	8.81	10
	IPFP	4.18	2.29	69.45	13.69	15.19	10.02	11.5

Table 2: **RMSE Comparison:** The top-3 lowest RMSEs per dataset are highlighted in green shades, with darker shades denoting better RMSE. An asterisk (*) marks the better value when additional decimal places resolve ties after rounding to two decimal places.

Methods	AIDS	Linux	IMDB	ogbg-molhiv	ogbg-code2	ogbg-molpcba	Avg. Rank
GRAIL-MIX	0.80	≈ 1	≈ 1	0.20	0.12	0.12	1.83
GREED	0.58	0.79	0.17	0.23	0.09	0.21	2.17
ERIC	0.37	0.92	0.08	0.21	0.01	0.18	2.83
GEDGNN	0.35	0.85	0.07	0.57	0.01*	0.09	3.17

Table 3: **EMR Comparison**: The top-3 highest EMRs per dataset are highlighted in green shades, with darker shades denoting better EMRs. The EMR values for GRAIL are in App. (Table 9). We omit them here as GRAIL-MIX performs similarly across datasets. For a focused comparison, we only include the top-3 baselines from Table 2, since the remaining do not provide competitive performance. For ties after rounding to two decimals, an asterisk (*) marks the higher value. Values in (0.99, 1) are shown as ≈ 1.

5.1 EXPERIMENT SETUP

Gemini-1.5 Pro has been used for all experiments. Further details of the software and hardware environments and hyper-parameters used for GRAIL are listed in App. A.2.1.
Datasets: Table 7 in the appendix summarizes the datasets used in this study. A detailed description of the data semantics is included in App. A.3. While AIDS, Linux and IMDB are obtained from Morris et al. (2020), the other four datasets are made available by Hu et al. (2021).
Benchmark Algorithms: The recent baselines are listed in Table 1. From this set, we benchmark GRAIL against GREED Ranjan et al. (2022), GEDGNN Piao et al. (2023), ERIC Zhuo & Tan (2022), GRAPHEDX Jain et al. (2024) and H2MN Zhang et al. (2021). We omit SIMGNN, GRAPHOTSIM, GMN, GRAPHSIM, TAGSIM and GENN-A*, since they have been outperformed by the considered baselines of GREED, GRAPHEDX, GEDGNN and ERIC.
Among non-neural baselines we include the best-performing heuristics from the benchmarking study in Blumenthal et al. (2020): namely, LP-GED-F2, COMPACT-MIP, ADJ-IP, BRANCH-TIGHT, NODE and IPFP.

GRAIL-MIX is a variant of GRAIL trained on a mixture of graph pairs from multiple datasets, while maintaining the same training set size as GRAIL. The programs discovered by GRAIL-MIX are used for inference across all datasets to assess whether a single training instance can generalize across domains, eliminating dataset-specific training.

Test set \Train Set	**AIDS**		**ogbg-molhiv**		**ogbg-molpcba**	
	GREED	GRAIL	GREED	GRAIL	GREED	GRAIL
AIDS	0.61	0.57	5.71	0.64	4.58	0.59
ogbg-molhiv	NA	3.02	3.02	2.96	3.86	2.89
ogbg-molpcba	NA	3.59	2.16	3.54	2.48	3.18

Table 4: **Intra-Domain Generalizability:** RMSE of the best neural method, GREED, and GRAIL. Off-diagonal entries represent cross-dataset performance. NA symbolizes that it's not possible to train a single model covering the train-test combination.

Train-Validation-Test Split: To construct the test set for a particular dataset, we select 1000 graph pairs uniformly at random and compute their true GED. The procedure for computing the ground truth GED is discussed in App. A.3.1. The training and validation sets depend on the algorithm.

- **Neural Algorithms:** All neural approaches are trained on 10,000 graph pairs per dataset. This training time exceeds 15 days for certain datasets (see Fig. 6a).
- **GRAIL and GRAIL-MIX:** GRAIL is trained with only 1,000 graph pairs per dataset. As discussed already, GRAIL does not require ground-truth GED. In GRAIL-MIX, we choose 166 graph pairs from each of the datasets listed in Table 7 except ogbg-ppa. Both GRAIL and GRAIL-MIX do not use a validation set.
- **Non-Neural Baselines:** These unsupervised algorithms do not require any training or validation datasets.

Test set \Train set	**AIDS**	**IMDB**	**Linux**	**ogbg-molhiv**	**ogbg-code2**	**ogbg-molpcba**
AIDS	0.57	0.63	0.65	0.64	0.62	0.59
IMDB	0.88	0.55	0.88	0.78	0.74	0.87
Linux	0.18	0.22	0.13	0.24	0.16	0.24
ogbg-molhiv	3.02	2.93	3.08	2.96	2.96	2.89
ogbg-code2	4.44	4.32	4.74	4.07	4.22	4.5
ogbg-molpcba	3.59	3.63	3.61	3.54	3.64	3.18

Table 5: **Inter-Domain Generalizability:** Generalization of GRAIL across domains, dataset sizes, and node label distributions by training on one dataset and measuring RMSE on others. Off-diagonal entries represent cross-dataset performance. The best two results for each test set(row) have been highlighted in shades of green, with darker being better.

Metrics: We employ two metrics: Root Mean Squared Error (*RMSE*) and Exact Match Ratio (*EMR*). *EMR* quantifies the proportion of test graph pairs for which the predicted GED exactly matches the true GED. (See App. A.3.2 for details.)

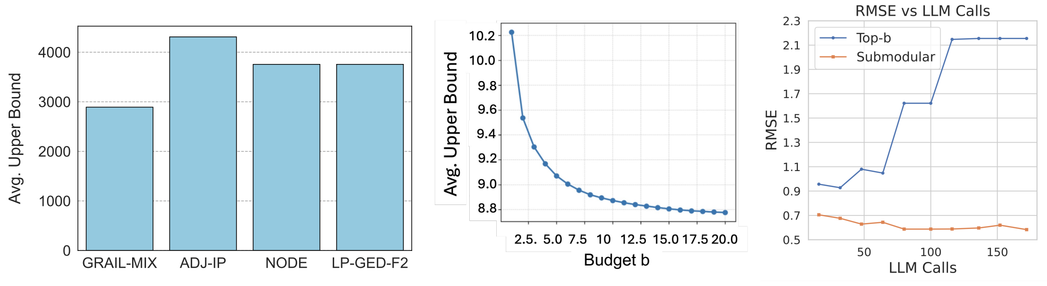
5.2 EMPIRICAL ANALYSIS OF APPROXIMATION ERRORS

Tables 2 and 3 benchmark GRAIL in terms of RMSE and EMR. Several important observations emerge. First, GRAIL and GRAIL-MIX comprehensively outperform the baselines despite not using any ground truth for training. This is a critical advantage as it saves time in expensive ground truth computation (Refer to Fig. 6a). The improvement is the highest in the IMDB dataset, which we specifically analyze in § A.4. Second, the efficacy of GRAIL-MIX across datasets demonstrates that the discovered programs are universally applicable on multiple datasets and can be called as *foundation functions*. These foundation functions eliminate the need for dataset-specific training. Third, GRAIL-MIX outperforms GRAIL in three datasets, indicating positive cross-dataset knowledge transfer.

5.3 GENERALIZABILITY

An intrinsic requirement of all machine learning methods is that the training and test data are sampled from the same distribution. Thus, the neural baselines depend on training data tailored to the test dataset, limiting their ability to transfer knowledge due to reliance on dataset-specific features. In contrast, GRAIL learns symbolic logical rules in the form of programs, facilitating out-of-domain and out-of-distribution generalization. We now evaluate this capability.

Intra-domain: Neural models are limited by feature dimensionality, making zero-shot generalization



(a) ogbg-ppa (b) AIDS (c) AIDS

Figure 3: (a) **GRAIL-MIX at scale:** Performance of GRAIL-MIX on the ogbg-ppa dataset when compared to the top-3 non-neural baselines on the basis of average rank in Table 2. (b) Impact of function budget on upper bound. (c) Impact of greedy submodular optimization on performance on test set.

infeasible. However, for domains such as chemical compounds, a uniform feature space allows training a single model. We retrain the best neural baseline, GREED, and compare its intra-domain generalizability with GRAIL in Table 4. Note that the AIDS dataset has a smaller feature dimension than the ogbg datasets, making it impossible to derive a common feature space. GRAIL generalizes well across all train-test dataset pairs, while GREED struggles except for ogbg-molhiv and ogbg-molpcba. This is because both datasets are adopted from the same parent dataset MoleculeNet Wu et al. (2018); Hu et al. (2020) with similar topological features (Table 7).

Inter-domain: Table 5 showcases the ability of the functions discovered by dataset-specific training of GRAIL to generalize across other datasets. We do not observe a significant increase in RMSE on the off-diagonal entries, which showcases positive knowledge transfer and an ability not seen in neural approximators.

Generalization to graph size: In this experiment, we evaluate GRAIL-MIX on the ogbg-ppa dataset, which has been omitted from prior benchmarking due to the large size of its graphs (See Table 7). Given the computational infeasibility of ground-truth GED computation for these graphs, neural approximators cannot be trained on this dataset. To assess the generalization capability of GRAIL-MIX, we compare the upper bound provided by its programs against the top-3 non-neural heuristics (based on Table 2). We observe that GRAIL-MIX provides 30% to 45% tighter upper bounds. Fig. 10 in the appendix further substantiates that GRAIL generalizes better to large graphs than neural baselines.

5.4 ABLATION STUDY AND PARAMETERS

Impact of budget b: For the true GED, all possible mappings (factorial in the graph size) must be considered. Instead, GRAIL restricts this to b mappings, where each mapping is generated by a program. In Fig. 3b, we plot how b affects the upper bound. As shown, the upper bound converges at ≈ 15 functions. Similar trends are observed in other datasets (see Fig. 5).

Impact of Submodularity: What happens if, instead of selecting the top-b functions using greedy submodular optimization, we evolve and score functions individually based on their upper bounds (Eq. 3) and select the top-b solely based on this criterion? Fig. 3c illustrates the impact on RMSE across training iterations (LLM calls). While selecting the top-b functions through submodular optimization shows a clear trend of decreasing RMSE on the test set, independently choosing the top-b functions based on individual scores results in significantly higher RMSE, with a progressive worsening trend indicative of overfitting to the training set. This result underscores the importance of submodularity in selecting functions that complement one another and perform well collectively (See Fig. 11 for additional metrics).

Interpretability: We present a case study in App. A.4 to showcase the interpretability aspects of GRAIL for the IMDB dataset, where it outperforms all neural baselines by a large margin (Table 2).

6 CONCLUSIONS AND FUTURE DIRECTIONS

This paper introduced a new paradigm of computing GED by leveraging LLMs to autonomously generate programs. Unlike traditional methods that rely on neural networks and require computationally expensive, NP-hard ground truth data, our method employs a self-evolutionary strategy to discover programs without any ground truth data. Remarkably, these programs not only surpass state-of-the-art methods on average but are also interpretable and demonstrate strong transferability across various datasets and domains. While our approach is demonstrated on GED computation, we believe it is generalizable to other combinatorial problems with similar constraints, both within and beyond graph-related tasks. An interesting direction for future work is to critically analyze the programs discovered by our method with domain experts and to develop mechanisms that facilitate closer cooperation between human and LLM agents.

ACKNOWLEDGEMENTS

The authors extend our sincere gratitude to Alexander Novikov (Research scientist, Google Deep-Mind) for their thorough review of the initial draft of this work. Their insightful comments and constructive feedback were instrumental in refining the manuscript and enhancing its clarity.

STATEMENT OF IMPACT

This work introduces a novel approach to Graph Edit Distance (GED) computation by leveraging large language models (LLMs) and evolutionary algorithms to generate interpretable programs for GED approximation. Unlike existing methods, our approach prioritizes transparency, interpretability, and cross-domain generalization while achieving state-of-the-art performance in approximation accuracy.

The societal implications of this work are significant. By addressing key limitations of neural approaches—such as their reliance on costly ground truth data, lack of interpretability, and domain-specific retraining—our method has the potential to make graph similarity computation more accessible and efficient across a variety of applications, including bioinformatics, social network analysis, and cheminformatics. Moreover, the transparency of program-based solutions could foster trust and reliability in critical domains where understanding the computation process is essential, such as healthcare or legal systems.

We do not foresee any ethical concerns arising out of our work.

REFERENCES

Jiyang Bai and Peixiang Zhao. Tagsim: type-aware graph similarity learning and computation. *Proc. VLDB Endow.*, 15(2):335–347, October 2021. ISSN 2150-8097. doi: 10.14778/3489496.3489513. URL https://doi.org/10.14778/3489496.3489513.

Yunsheng Bai, Hao Ding, Song Bian, Ting Chen, Yizhou Sun, and Wei Wang. Simgnn: A neural network approach to fast graph similarity computation. In *WSDM*, WSDM '19, pp. 384–392, 2019.

Yunsheng Bai, Hao Ding, Ken Gu, Yizhou Sun, and Wei Wang. Learning-based efficient graph similarity computation via multi-scale convolutional set matching. *AAAI*, pp. 3219–3226, Apr. 2020.

David B. Blumenthal. New techniques for graph edit distance computation. *CoRR*, abs/1908.00265, 2019. URL http://arxiv.org/abs/1908.00265.

David B. Blumenthal and Johann Gamper. Improved lower bounds for graph edit distance. *IEEE Transactions on Knowledge and Data Engineering*, 30(3):503–516, 2018. doi: 10.1109/TKDE.2017.2772243.

David B. Blumenthal and Johann Gamper. On the exact computation of the graph edit distance. *Pattern Recogn. Lett.*, 134(C):46–57, June 2020. ISSN 0167-8655. doi: 10.1016/j.patrec.2018.05.002. URL https://doi.org/10.1016/j.patrec.2018.05.002.

David B Blumenthal, Nicolas Boria, Johann Gamper, Sébastien Bougleux, and Luc Brun. Comparing heuristics for graph edit distance computation. *The VLDB journal*, 29(1):419–458, 2020.

Aditya Bommakanti, Harshith Reddy Vonteri, Sayan Ranu, and Panagiotis Karras. Eugene: Explainable unsupervised approximation of graph edit distance, 2024a. URL https://arxiv.org/abs/2402.05885.

Aditya Bommakanti, Harshith Reddy Vonteri, Konstantinos Skitsas, Sayan Ranu, Davide Mottin, and Panagiotis Karras. Fugal: Feature-fortified unrestricted graph alignment. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024b.

Jiazhou Chen, Hong Peng, Guoqiang Han, Hongmin Cai, and Jiulun Cai. Hogmmnc: a higher order graph matching with multiple network constraints model for gene–drug regulatory modules identification. *Bioinformatics*, 35(4):602–610, 2019.

Donatello Conte, Pasquale Foggia, Carlo Sansone, and Mario Vento. Graph matching applications in pattern recognition and image processing. *Proceedings 2003 International Conference on Image Processing (Cat. No.03CH37429)*, 2:II–21, 2003. URL `https://api.semanticscholar.org/CorpusID:267842199`.

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009. ISBN 0262033844.

Khoa D. Doan, Saurav Manchanda, Suchismit Mahapatra, and Chandan K. Reddy. Interpretable graph similarity computation via differentiable optimal alignment of node embeddings. In *SIGIR*, pp. 665–674, 2021.

Zhou Fan, Cheng Mao, Yihong Wu, and Jiaming Xu. Spectral graph matching and regularized quadratic relaxations: Algorithm and theory. In Hal Daumé III and Aarti Singh (eds.), *Proceedings of the 37th International Conference on Machine Learning*, volume 119 of *Proceedings of Machine Learning Research*, pp. 2985–2995. PMLR, 13–18 Jul 2020.

Francesco Di Giovanni, T. Konstantin Rusch, Michael Bronstein, Andreea Deac, Marc Lackenby, Siddhartha Mishra, and Petar Veličković. How does over-squashing affect the power of GNNs? *Transactions on Machine Learning Research*, 2024. ISSN 2835-8856. URL `https://openreview.net/forum?id=KJRoQvRWNs`.

V. Scott Gordon and L. Darrell Whitley. Serial and parallel genetic algorithms as function optimizers. In *Proceedings of the 5th International Conference on Genetic Algorithms*, pp. 177–183, San Francisco, CA, USA, 1993. Morgan Kaufmann Publishers Inc. ISBN 1558602992.

Huahai He and A.K. Singh. Closure-tree: An index structure for graph queries. In *22nd International Conference on Data Engineering (ICDE'06)*, pp. 38–38, 2006. doi: 10.1109/ICDE.2006.37.

John E Hopcroft and Richard M Karp. An n^5/2 algorithm for maximum matchings in bipartite graphs. *SIAM Journal on Computing*, 2(4):225–231, 1973.

Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. Open graph benchmark: Datasets for machine learning on graphs. In H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin (eds.), *Advances in Neural Information Processing Systems*, volume 33, pp. 22118–22133. Curran Associates, Inc., 2020.

Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. Open graph benchmark: Datasets for machine learning on graphs, 2021. URL `https://arxiv.org/abs/2005.00687`.

Eeshaan Jain, Indradyumna Roy, Saswat Meher, Soumen Chakrabarti, and Abir De. Graph edit distance with general costs using neural set divergence. In *The Thirty-eighth Annual Conference on Neural Information Processing Systems*, 2024.

D. Justice and A. Hero. A binary linear programming formulation of the graph edit distance. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 28(8):1200–1214, 2006. doi: 10.1109/TPAMI.2006.152.

Harold W Kuhn. The hungarian method for the assignment problem. *Naval Research Logistics Quarterly*, 2(1-2):83–97, 1955.

Marius Leordeanu, Martial Hebert, and Rahul Sukthankar. An integer projected fixed point method for graph matching and map inference. *Advances in neural information processing systems*, 22, 2009.

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

Julien Lerouge, Zeina Abu-Aisheh, Romain Raveaux, Pierre Héroux, and Sébastien Adam. New binary linear programming formulation to compute the graph edit distance. *Pattern Recognition*, 72:254–265, 2017b. ISSN 0031-3203. doi: https://doi.org/10.1016/j.patcog.2017.07.029. URL `https://www.sciencedirect.com/science/article/pii/S003132031730300X`.

Yujia Li, Chenjie Gu, Thomas Dullien, Oriol Vinyals, and Pushmeet Kohli. Graph matching networks for learning the similarity of graph structured objects. In *ICML*, pp. 3835–3845, 2019.

Christopher Morris, Nils M. Kriege, Franka Bause, Kristian Kersting, Petra Mutzel, and Marion Neumann. Tudataset: A collection of benchmark datasets for learning with graphs. *CoRR*, abs/2007.08663, 2020. URL `https://arxiv.org/abs/2007.08663`.

Chengzhi Piao, Tingyang Xu, Xiangguo Sun, Yu Rong, Kangfei Zhao, and Hong Cheng. Computing graph edit distance via neural graph matching. *Proceedings of the VLDB Endowment*, 16(8): 1817–1829, 2023.

Rishabh Ranjan, Siddharth Grover, Sourav Medya, Venkatesan Chakaravarthy, Yogish Sabharwal, and Sayan Ranu. Greed: A neural framework for learning graph distance functions. *Advances in Neural Information Processing Systems*, 35:22518–22530, 2022.

Sayan Ranu, Minh Hoang, and Ambuj Singh. Answering top-k representative queries on graph databases. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pp. 1163–1174, 2014.

Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Matej Balog, M Pawan Kumar, Emilien Dupont, Francisco JR Ruiz, Jordan S Ellenberg, Pengming Wang, Omar Fawzi, et al. Mathematical discoveries from program search with large language models. *Nature*, 625(7995):468–475, 2024.

Rohit Singh, Jinbo Xu, and Bonnie Berger. Global alignment of multiple protein interaction networks with application to functional orthology detection. *Proceedings of the National Academy of Sciences*, 105(35):12763–12768, 2008. doi: 10.1073/pnas.0806627105. URL `https://www.pnas.org/doi/abs/10.1073/pnas.0806627105`.

Damian Szklarczyk, Annika L Gable, David Lyon, Alexander Junge, Stefan Wyder, Jaime Huerta-Cepas, Milan Simonovic, Nadezhda T Doncheva, John H Morris, and Peer Bork et al. String v11: protein–protein association networks with increased coverage, supporting functional discovery in genome-wide experimental datasets. *Nucleic Acids Research*, 47(D1):D607–D613, 2019.

Runzhong Wang, Tianqi Zhang, Tianshu Yu, Junchi Yan, and Xiaokang Yang. Combinatorial learning of graph edit distance via dynamic embedding. In *IEEE Conference on Computer Vision and Pattern Recognition*, 2021.

Xiaoli Wang, Xiaofeng Ding, Anthony K. H. Tung, Shanshan Ying, and Hai Jin. An efficient graph indexing method. In *Proceedings of the 2012 IEEE 28th International Conference on Data Engineering (ICDE '12)*, pp. 210–221, USA, 2012. IEEE Computer Society.

Zhenqin Wu, Bharath Ramsundar, Evan N Feinberg, Joseph Gomes, Caleb Geniesse, Aneesh S Pappu, Karl Leswing, and Vijay Pande. Moleculenet: a benchmark for molecular machine learning. *Chemical science*, 9(2):513–530, 2018.

Pinar Yanardag and S.V.N. Vishwanathan. Deep graph kernels. In *Proceedings of the 21st ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '15)*, pp. 1365–1374, New York, NY, USA, 2015. Association for Computing Machinery.

Zhen Zhang, Jiajun Bu, Martin Ester, Zhao Li, Chengwei Yao, Zhi Yu, and Can Wang. H2mn: Graph similarity learning with hierarchical hypergraph matching networks. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*, KDD '21, pp. 2274–2284, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450383325. doi: 10.1145/3447548.3467328. URL `https://doi.org/10.1145/3447548.3467328`.

Wei Zhuo and Guang Tan. Efficient graph similarity computation with alignment regularization. *Advances in Neural Information Processing Systems*, 35:30181–30193, 2022.

Marinka Zitnik, Marcus W Feldman, and Jure Leskovec et al. Evolution of resilience in protein interactomes across the tree of life. *Proceedings of the National Academy of Sciences*, 116(10): 4426–4433, 2019.

A APPENDIX

A.1 PROOFS

A.1.1 NP-HARDNESS OF EQ. 4

REDUCTION TO PROVE NP-HARDNESS

We reduce the *Set Cover* problem to the given problem in polynomial time to demonstrate its NP-hardness.

Given a universe of elements $U = \{e_1, e_2, \dots, e_n\}$, a collection of sets $\mathcal{S} = \{S_1, S_2, \dots, S_m\}$ where $S_i \subseteq U$, and a budget b, the set cover problem seeks to determine if there exist b sets $S_1, \dots, S_b \in \mathcal{S}$ whose union covers all elements of U.

Given an instance of the set cover problem, we construct a bipartite graph $\mathcal{B} = (\mathcal{V}, \mathcal{U}, \mathcal{E}, \mathcal{W})$, where $\mathcal{V} = \mathcal{S}, \mathcal{U} = U$, and an edge $(S_i, e_j) \in \mathcal{E}$ exists if and only if S_i covers e_j. Each edge (S_i, e_j) has a weight:

$$w(S_i, e_j) = \begin{cases} 1 & \text{if } S_i \text{ covers } e_j, \\ 1 + \Delta & \text{if } S_i \text{ does not cover } e_j. \end{cases}$$

where $\Delta > 0$.

The objective is to select b nodes from $\mathcal{V}$ (representing sets $\mathcal{S}$) such that Eq. 4 is minimized on graph $\mathcal{B}$.

If a *Set Cover* of size b exists, then all n elements can be covered by b sets. This means if we select the corresponding nodes $\mathcal{A}^* \subseteq \mathcal{V}$, then every node in $\mathcal{U}$ will have at least one edge of weight 1 from some node in $\mathcal{A}^*$ incident on it. Hence, $\mathcal{J}(\mathcal{A}^*)$ will return a cumulative sum of n.

Conversely, if no *Set Cover* of size b exists, then some elements will not be covered by the selected sets, and their corresponding nodes in $\mathcal{U}$ will have only edges of edge weights $1 + \Delta$ from nodes in $\mathcal{A}^*$.

Therefore, a solution to the *Set Cover* problem exists iff selecting the corresponding nodes $\mathcal{A}^* \subseteq \mathcal{V}$ leads to $\mathcal{J}(\mathcal{A}^*) = n$. Conversely, if $\mathcal{J}(\mathcal{A}^*) > n$, it implies that no b-set cover exists.

A.1.2 MONOTONICITY AND SUBMODULARITY

Lemma 1. *Monotonicity:* $\mathcal{J}(\mathcal{A}) \leq \mathcal{J}(\mathcal{A}')$ *if* $\mathcal{A} \supseteq \mathcal{A}'$.

Proof. Since $\mathcal{J}(\mathcal{A}')$ computes minimum over all available mappings in $\mathcal{A}'$, the minimum can only reduce when additional mappings are added to form $\mathcal{A}$. □

Lemma 2. *Submodularity:* $\mathcal{J}(\mathcal{A} \cup \{P\}) - \mathcal{J}(\mathcal{A}) \leq \mathcal{J}(\mathcal{A}' \cup \{P\}) - \mathcal{J}(\mathcal{A}')$.

Proof. We seek to show that the marginal reduction in $\mathcal{J}(\mathcal{A})$ when a program (mapping) P is added to $\mathcal{A}$ is atmost as large as adding P to its subset $\mathcal{A}'$. We establish this through *proof by contradiction*.

Let us assume

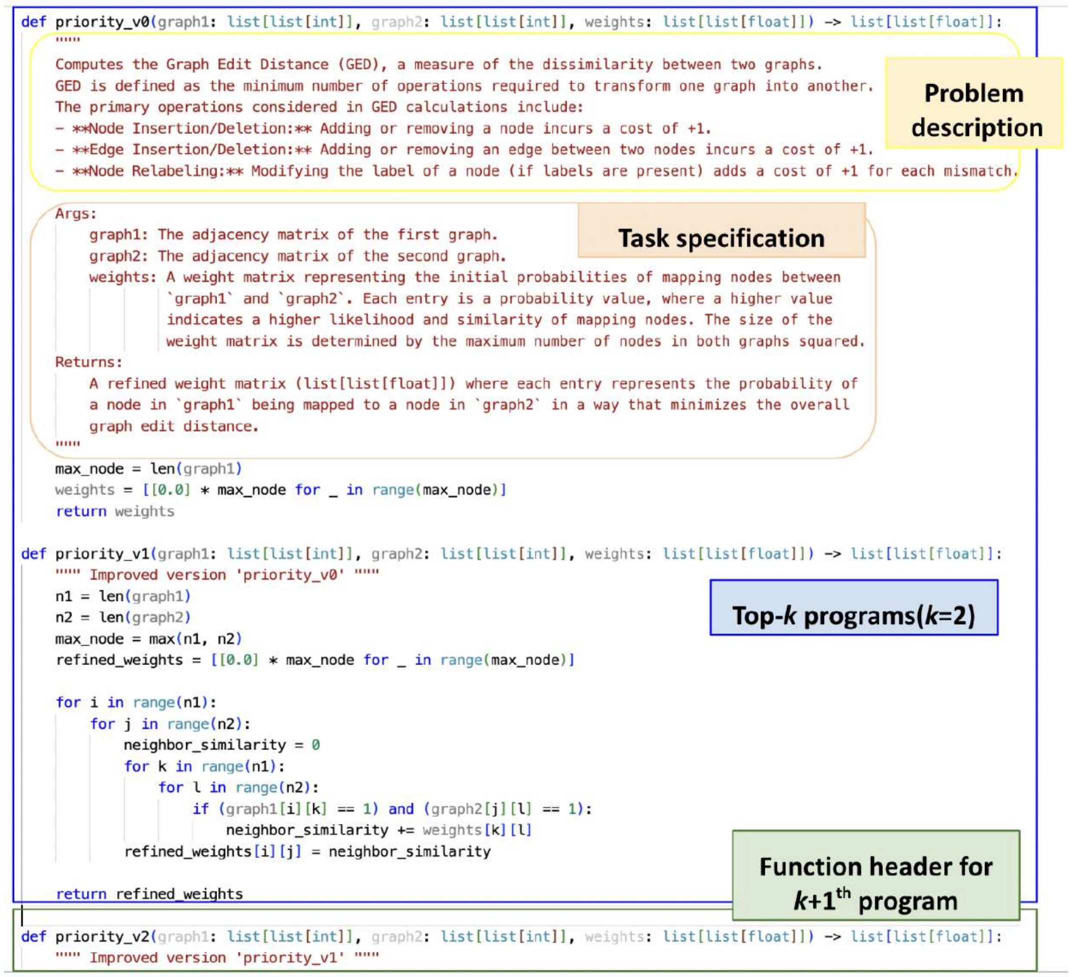
$$\exists \mathcal{A} \supseteq \mathcal{A}',\ \mathcal{J}(\mathcal{A} \cup \{P\}) - \mathcal{J}(\mathcal{A}) > \mathcal{J}(\mathcal{A}' \cup \{P\}) - \mathcal{J}(\mathcal{A}') \tag{7}$$

Due to the min operator in Eq. 5, Eq. 7 implies that the additional number of graph pairs where P contributes to the minimum mapping is higher when added to $\mathcal{A}$ than when added to $\mathcal{A}'$. This creates a contradiction, since if P contributes to the minimum of a graph pair in $\mathcal{A} \cup \{P\}$, then it guaranteed to contribute to the minimum for the same pair in $\mathcal{A}' \cup \{P\}$ as well. □

A.2 EXPERIMENTS

A.2.1 SETUP

All experiments ran on a machine equipped with an Intel Xeon Gold 6142 CPU @1GHz and a GeForce GTX 1080 Ti GPU. While non-neural methods and GRAIL run on the CPU, neural baselines



```python
def priority_v0(graph1: list[list[int]], graph2: list[list[int]], weights: list[list[float]]) -> list[list[float]]:
    """
    Computes the Graph Edit Distance (GED), a measure of the dissimilarity between two graphs.
    GED is defined as the minimum number of operations required to transform one graph into another.
    The primary operations considered in GED calculations include:
    - **Node Insertion/Deletion:** Adding or removing a node incurs a cost of +1.
    - **Edge Insertion/Deletion:** Adding or removing an edge between two nodes incurs a cost of +1.
    - **Node Relabeling:** Modifying the label of a node (if labels are present) adds a cost of +1 for each mismatch.

    Args:
        graph1: The adjacency matrix of the first graph.
        graph2: The adjacency matrix of the second graph.
        weights: A weight matrix representing the initial probabilities of mapping nodes between
                 `graph1` and `graph2`. Each entry is a probability value, where a higher value
                 indicates a higher likelihood and similarity of mapping nodes. The size of the
                 weight matrix is determined by the maximum number of nodes in both graphs squared.
    Returns:
        A refined weight matrix (list[list[float]]) where each entry represents the probability of
        a node in `graph1` being mapped to a node in `graph2` in a way that minimizes the overall
        graph edit distance.
    """
    max_node = len(graph1)
    weights = [[0.0] * max_node for _ in range(max_node)]
    return weights

def priority_v1(graph1: list[list[int]], graph2: list[list[int]], weights: list[list[float]]) -> list[list[float]]:
    """ Improved version 'priority_v0' """
    n1 = len(graph1)
    n2 = len(graph2)
    max_node = max(n1, n2)
    refined_weights = [[0.0] * max_node for _ in range(max_node)]

    for i in range(n1):
        for j in range(n2):
            neighbor_similarity = 0
            for k in range(n1):
                for l in range(n2):
                    if (graph1[i][k] == 1) and (graph2[j][l] == 1):
                        neighbor_similarity += weights[k][l]
            refined_weights[i][j] = neighbor_similarity

    return refined_weights

def priority_v2(graph1: list[list[int]], graph2: list[list[int]], weights: list[list[float]]) -> list[list[float]]:
    """ Improved version 'priority_v1' """
```

Figure 4: Example of an input prompt to GRAIL

exploit the GPU. For the LLM, we use Gemini 1.5 Pro. In particular, we have used the initial stable version of Gemini 1.5 Pro, i.e., gemini-1.5-pro-001, which was released on May 24, 2024.

Hyper-parameters: Table H lists the hyper-parameters used for GRAIL. k stands for the number of

Hyper-parameter	Value
k	2
b	15
number of islands	5
temperature	0.99
Algorithm for bipartite matching	Neighbor-biased mapper He & Singh (2006)

Table H: Hyper-parameters used for GRAIL

functions per response generated by the LLM and b is the function budget employed for submodularity while training. We decided to use k as 2, since with greater values of k, we observed no significant improvement in quality metrics. This was also observed in FunSearch Romera-Paredes et al. (2024). For the function budget b, we observed that a value of 15 was good enough for most datasets (Refer Fig. 5).

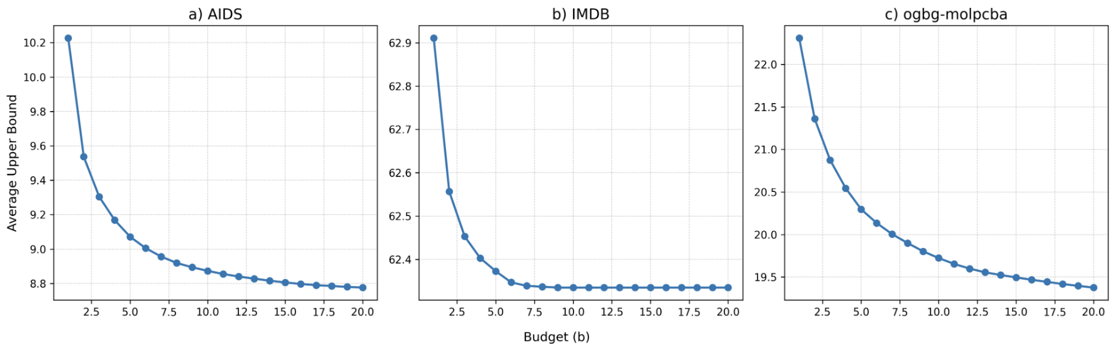


Figure 5: Avg. Upper Bound vs function budget (b) for submodular greedy selection

Name	**# Graphs**	**Avg $\|V\|$**	**Avg $\|E\|$**	**# labels**	**Domain**
ogbg-molhiv	39650	24	52	119	Molecules
ogbg-molpcba	436313	26	56	119	Molecules
ogbg-code2	139468	37	72	97	Software
AIDS	700	9	9	29	Molecules
Linux	1000	8	7	Unlabeled	Software
IMDB	1500	13	65	Unlabeled	Movies
ogbg-ppa	39650	243.4	2226.1	Unlabeled	Protein

Table 7: Datasets used for benchmarking GRAIL.

A.3 DATASETS

The semantics of these datasets are as follows:

- **ogbg-molhiv and ogbg-molpcba:** These are chemical compound datasets, with each graph representing a molecule. Nodes in these graphs correspond to atoms and are labeled with their atomic numbers, while edges denote the chemical bonds between atoms. These datasets vary in size and complexity, with a rich diversity of molecular structures, enabling us to test the robustness and generalizability of our method.
- **ogbg-code2:** This dataset comprises a vast collection of Abstract Syntax Trees (ASTs) generated from nearly 450,000 Python method definitions. Each graph in this dataset represents an AST, with nodes labeled from a predefined set of 97 categories, capturing various syntactic constructs within the methods. These graphs are considered undirected, simplifying the representation while preserving structural relationships.
- **ogbg-ppa:** This dataset includes undirected protein association neighborhoods extracted from protein-protein interaction networks of 1,581 speciesSzklarczyk et al. (2019) across 37 diverse taxonomic groups. To build these neighborhoods, 100 proteins were randomly selected from each species, and 2-hop protein association neighborhoods were constructed around each selected proteinZitnik et al. (2019). In these graphs, proteins are represented as nodes, and edges indicate biologically relevant associations between them.
- **AIDS:** This dataset is a collection of graphs sourced from the AIDS antiviral screen database, each graph representing a chemical compound's molecular structure. These graphs are labeled, capturing meaningful properties of the compounds, and are compact in size, containing no more than 10 nodes.
- **Linux:** A collection of program dependence graphs where nodes correspond to statements and edges indicate dependencies between statements. The graph sizes in this dataset are also limited to 10 nodes. This dataset is unlabeled and was introduced in Wang et al. (2012).
- **IMDB:** This dataset consists of ego-networks of actors and actresses who have shared screen time in movies. Each graph represents an ego-network where the nodes correspond to individuals (actors or actresses), and the edges denote shared appearances in films. This dataset is unlabeled and was introduced in Yanardag & Vishwanathan (2015).

General dataset statistics have been provided in Table 7.

A.3.1 GROUND-TRUTH DATA GENERATION

We employ MIP-F2 Lerouge et al. (2017b) to generate ground truth GED. MIP-F2 returns the lower and upper bounds of GED. We compute these bounds with a time limit of 600 seconds per pair. Pairs with equal lower and upper bounds are included in the ground truth.

A.3.2 METRICS

We use the following two metrics to quantify accuracy:

- **RMSE:** Evaluates the prediction accuracy by measuring the disparities between actual and predicted values. For n graph pairs, it is defined as:

$$\sqrt{\frac{1}{n}\sum_{i=1}^{n}(\text{true-ged}_i - \text{pred-ged}_i)^2}$$

- **Exact Match Ratio:** Represents the proportion of graph pairs where the predicted GED exactly matches the actual GED. For n graph pairs, it is defined as:

$$\frac{1}{n}\sum_{i=1}^{n}\mathbb{I}\big(\text{true-ged}_i = \text{pred-ged}_i\big)$$

  where $\mathbb{I}(\cdot)$ is an indicator function that returns 1 if the condition inside is true, and 0 otherwise. A higher Exact Match Ratio indicates better predictive accuracy at the individual graph pair level.

A.3.3 TEST DATA:

The statistics of the test data used for the evaluation of GRAIL are presented in Table. 5.

Name	**# Graph pairs**	**Avg $\|V\|$**	**Avg $\|E\|$**
AIDS	1000	8.8	8.8
Linux	1000	7.6	7.0
IMDB	967	12.2	57.0
ogbg-molhiv	902	23.0	49.5
ogbg-molpcba	859	25.0	54.1
ogbg-code2	968	36.7	35.7

Table H: Test Data Statistics

Method	**AIDS**	**Linux**	**IMDB**	**ogbg-molhiv**	**ogbg-code2**	**ogbg-molpcba**
GRAIL	0.83	≈ 1	0.99	0.18	0.11	0.12

Table 9: EMR results of GRAIL for all datasets. Values in the range (0.99,1) are denoted as ≈ 1

A.3.4 EFFICIENCY ANALYSIS

The training and inference time analysis is shown in Fig. 6.

Training Time: From Fig. 6a, we observe that GRAIL is significantly more efficient than the neural baselines. Neural methods require NP-hard ground truth training data, which involves extensive computation times of up to 15 days. Additionally, note that GRAIL-MIX requires training only once while performing on par with GRAIL and neural baselines in terms of approximation error (see Table 2).

Inference Time: We compare the inference time of GRAIL with the top three neural and non-neural methods from Table 2, as shown in Fig. 6b. At the onset, we point out that while GRAIL infers on CPUs and provides the node mapping in addition to the predicted GED, neural baselines rely on GPUs and only provide the GED. Hence, neural methods have a lower computational workload while having access to more powerful computational resources. Results indicate that GRAIL achieves faster inference times than neural baselines for smaller and sparser datasets, such as AIDS and Linux.

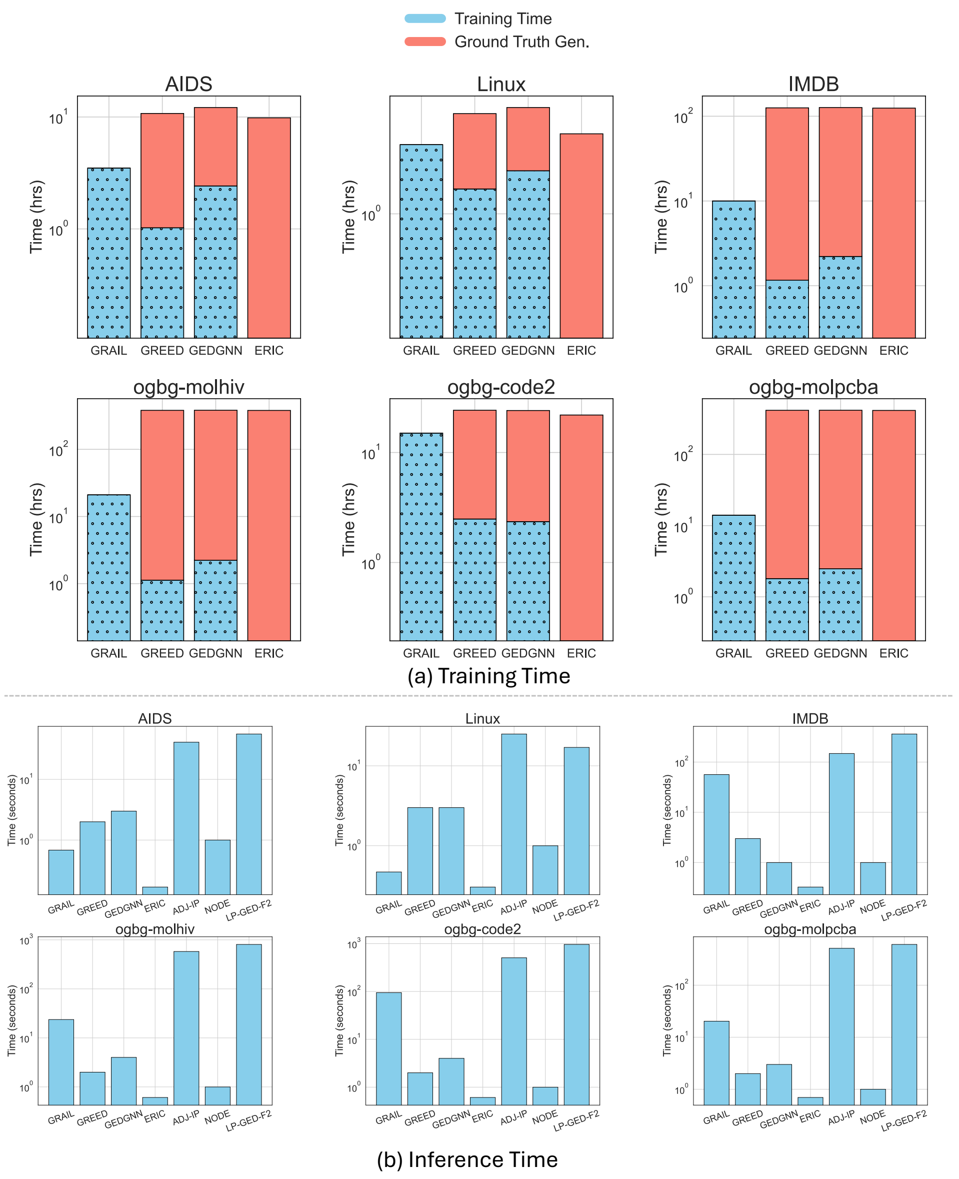


Figure 6: (a) Training Time: Comparison of GRAIL with the top-3 neural methods in Table 2. (b) Inference Time: Comparison of GRAIL with the top-3 neural and non-neural methods in Table 2. The top-3 methods have been selected based on avg. ranks.
Note: Ground truth generation time is the same for a dataset (9 hrs 43 min : AIDS, 3 hrs 25 min : Linux, 124 hrs 25 min : IMDB, 379 hrs 19 min : ogbg-molhiv, 21 hrs 42 min : ogbg-code2, 414 hrs 30 min : ogbg-molpcba) for all neural methods, but appears to be different in the plots due to log scale conversion.

However, inference times increase for larger and denser datasets, such as IMDB and ogbg, due to the computational overhead of computing mappings. The maximum recorded inference time is 94.6 seconds for 968 graph pairs in the ogbg-code2 dataset (~ 0.1 seconds per pair), which remains reasonable considering the various advantages of GRAIL, including its independence from ground truth data, one-time training for GRAIL-MIX, and strong generalization capabilities. Furthermore, the efficiency of GRAIL's programs can be further improved through human intervention or translation to more efficient languages, such as C.

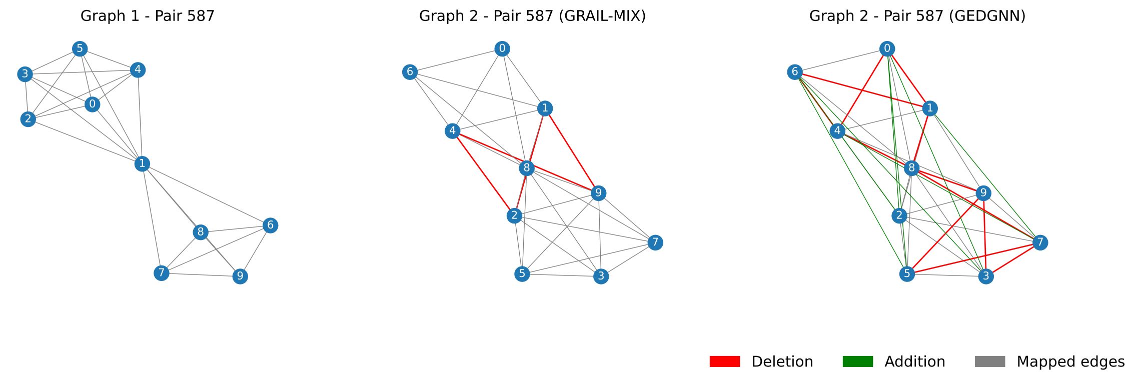


Figure 7: IMDB Case Study: The left-most graph represents Graph 1, while the middle and right-most graphs depict Graph 2 with predicted edits from GRAIL-MIX (Fig: 9) and GEDGNN, respectively. The red and green edges in each graph indicate the edge edits predicted by both methods. Ground Truth GED:4, GRAIL-MIX GED:4, GEDGNN Mapping's GED: 20.

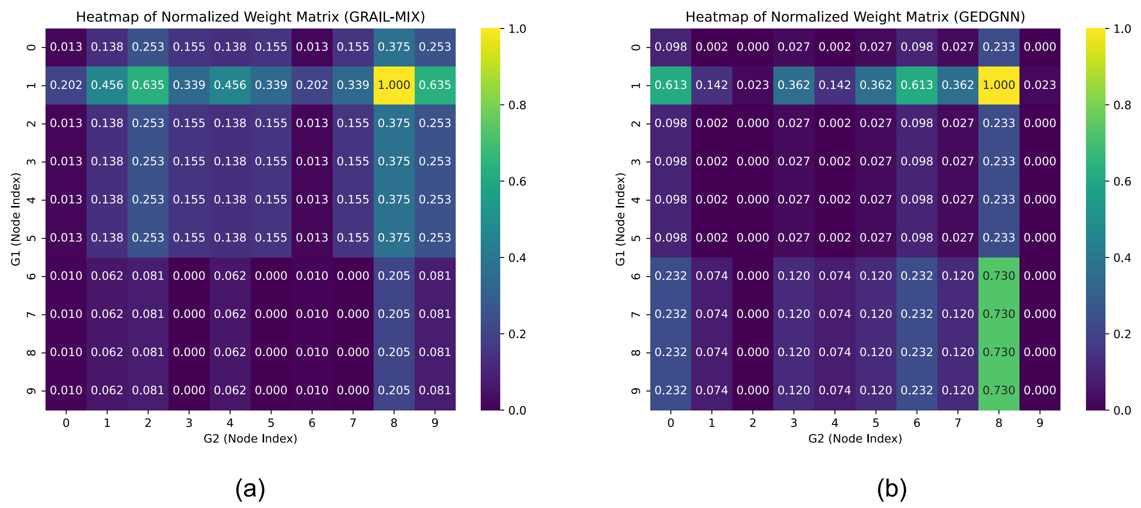
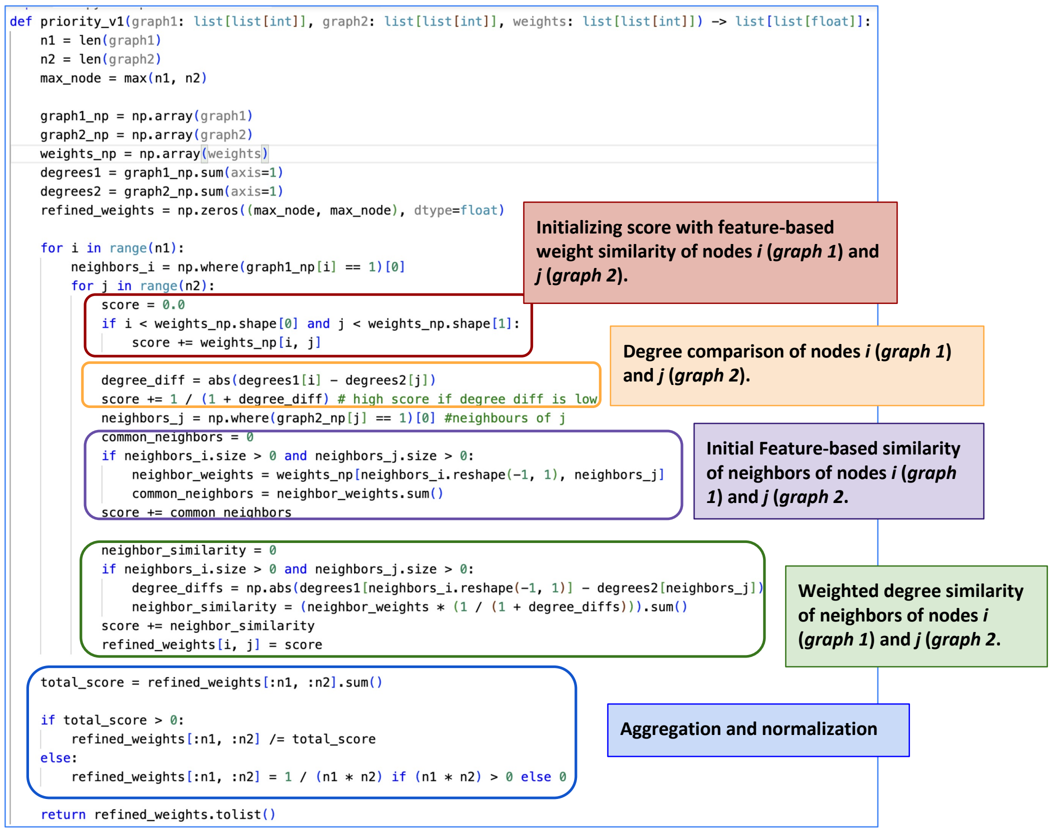


Figure 8: IMDB Case Study: Heatmap of weight matrix generated by (a) GRAIL-MIX (Fig: 9) and (b) GEDGNN

A.4 INTERPRETABILITY: CASE STUDY ON IMDB

To shed light on the superior performance of GRAIL over GNN-based neural approximators, we analyze a graph pair from the IMDB dataset, where GRAIL shows the highest improvement over all baselines (Table 2). Figures 7 and 8 in the Appendix illustrate the graph structures, the edits made by GRAIL and its closest competitor GEDGNN, and the node similarity matrices generated by these algorithms. The program discovered by GRAIL-MIX, shown in Fig. 9, achieves the ground truth GED of 4, while GEDGNN predicts a GED of 20. This program assigns node similarity scores based on degree similarity and that of their neighbors. Since IMDB is unlabeled, feature similarity does not influence the results.

Examining GEDGNN's similarity matrix reveals a different score distribution compared to GRAIL. For instance, node 1 in Graph 1 has the second-highest similarity to nodes 2 and 9 in GRAIL-MIX, but GEDGNN assigns low similarity to these nodes, favoring nodes 0 and 6 instead. GRAIL-MIX's decision aligns with the similarity in their degrees (degree of 9 for node 1 in Graph 1 versus 7 for nodes 2 and 9 in Graph 2). In contrast, GEDGNN, as a neural network, operates as a black box. We hypothesize that the poor performance of GEDGNN and other GNN-based algorithms in IMDB is due to the dataset's unlabeled nature and high density, leading to oversquashing. Giovanni et al. (2024).


Initializing score with feature-based weight similarity of nodes *i* (*graph 1*) and *j* (*graph 2*).
Degree comparison of nodes *i* (*graph 1*) and *j* (*graph 2*).
Initial Feature-based similarity of neighbors of nodes *i* (*graph 1*) and *j* (*graph 2*.
Weighted degree similarity of neighbors of nodes *i* (*graph 1*) and *j* (*graph 2*.
Aggregation and normalization

```python
def priority_v1(graph1: list[list[int]], graph2: list[list[int]], weights: list[list[int]]) -> list[list[float]]:
    n1 = len(graph1)
    n2 = len(graph2)
    max_node = max(n1, n2)

    graph1_np = np.array(graph1)
    graph2_np = np.array(graph2)
    weights_np = np.array(weights)
    degrees1 = graph1_np.sum(axis=1)
    degrees2 = graph2_np.sum(axis=1)
    refined_weights = np.zeros((max_node, max_node), dtype=float)

    for i in range(n1):
        neighbors_i = np.where(graph1_np[i] == 1)[0]
        for j in range(n2):
            score = 0.0
            if i < weights_np.shape[0] and j < weights_np.shape[1]:
                score += weights_np[i, j]

            degree_diff = abs(degrees1[i] - degrees2[j])
            score += 1 / (1 + degree_diff) # high score if degree diff is low
            neighbors_j = np.where(graph2_np[j] == 1)[0] #neighbours of j
            common_neighbors = 0
            if neighbors_i.size > 0 and neighbors_j.size > 0:
                neighbor_weights = weights_np[neighbors_i.reshape(-1, 1), neighbors_j]
                common_neighbors = neighbor_weights.sum()
            score += common_neighbors

            neighbor_similarity = 0
            if neighbors_i.size > 0 and neighbors_j.size > 0:
                degree_diffs = np.abs(degrees1[neighbors_i.reshape(-1, 1)] - degrees2[neighbors_j])
                neighbor_similarity = (neighbor_weights * (1 / (1 + degree_diffs))).sum()
            score += neighbor_similarity
            refined_weights[i, j] = score

    total_score = refined_weights[:n1, :n2].sum()

    if total_score > 0:
        refined_weights[:n1, :n2] /= total_score
    else:
        refined_weights[:n1, :n2] = 1 / (n1 * n2) if (n1 * n2) > 0 else 0

    return refined_weights.tolist()
```

Figure 9: IMDB Case Study: Program discovered by GRAIL-MIX that has minimum individual RMSE on IMDB dataset.

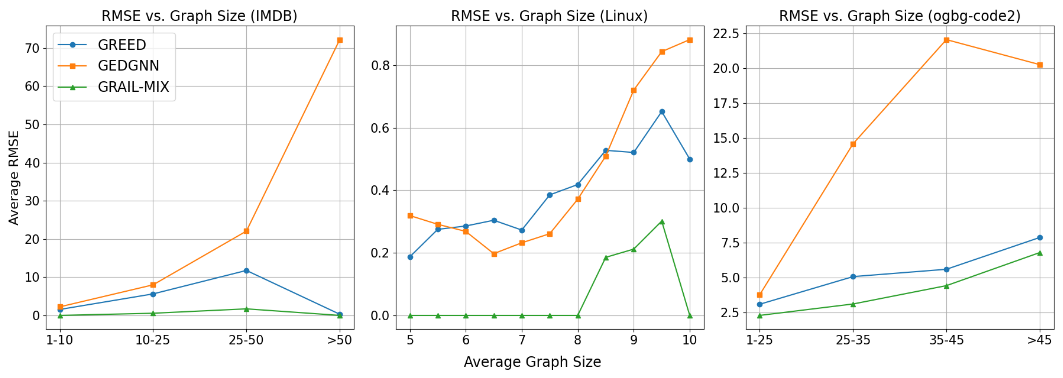


Figure 10: Avg. RMSE vs. Avg. Graph Size comparison on IMDB, Linux and ogbg-code2 datasets. GRAIL-MIX outperforms the best baselines at both smaller and larger graph sizes. The rate of increase of error is lower for GRAIL-MIX as opposed to GREED and GEDGNN with increasing average graph size.

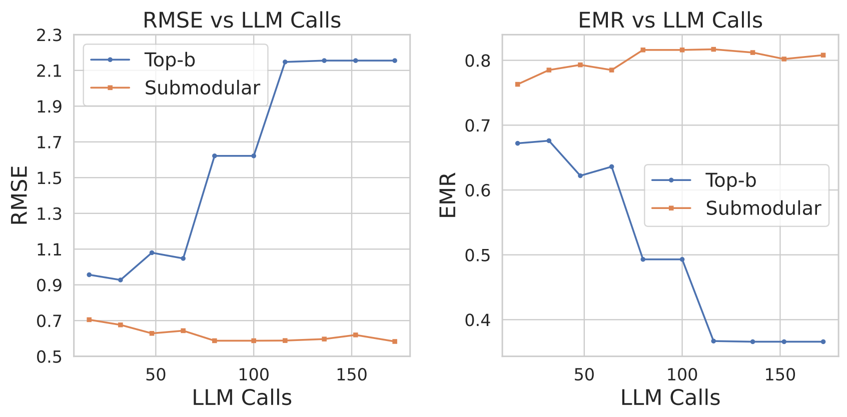


Figure 11: Performance comparison of Top-b vs. Greedy Submodular on the test set of AIDS dataset with an increasing number of LLM calls.