PL4XGL: A Programming Language Approach to Explainable Graph Learning

MINSEOK JEON, Korea University, Republic of Korea
JIHYEOK PARK, Korea University, Republic of Korea
HAKJOO OH, Korea University, Republic of Korea

In this article, we present a new, language-based approach to explainable graph learning. Though graph neural networks (GNNs) have shown impressive performance in various graph learning tasks, they have severe limitations in explainability, hindering their use in decision-critical applications. To address these limitations, several GNN explanation techniques have been proposed using a post-hoc explanation approach providing subgraphs as explanations for classification results. Unfortunately, however, they have two fundamental drawbacks in terms of 1) additional explanation costs and 2) the correctness of the explanations. This paper aims to address these problems by developing a new graph-learning method based on programming language techniques. Our key idea is two-fold: 1) designing a graph description language (GDL) to explain the classification results and 2) developing a new GDL-based interpretable classification model instead of GNN-based models. Our graph-learning model, called PL4XGL, consists of a set of candidate GDL programs with labels and quality scores. For a given graph component, it searches the best GDL program describing the component and provides the corresponding label as the classification result and the program as the explanation. In our approach, learning from data is formulated as a program-synthesis problem, and we present top-down and bottom-up algorithms for synthesizing GDL programs from training data. Evaluation using widely-used datasets demonstrates that PL4XGL produces high-quality explanations that outperform those produced by the state-of-the-art GNN explanation technique, SubgraphX. We also show that PL4XGL achieves competitive classification accuracy comparable to popular GNN models.

CCS Concepts: • **Software and its engineering** → *Domain specific languages.*

Additional Key Words and Phrases: Graph Learning, Domain-Specific Language, Program Synthesis

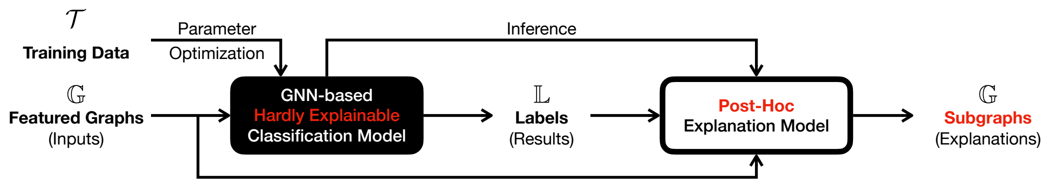
ACM Reference Format:
Minseok Jeon, Jihyeok Park, and Hakjoo Oh. 2024. PL4XGL: A Programming Language Approach to Explainable Graph Learning. *Proc. ACM Program. Lang.* 8, PLDI, Article 234 (June 2024), 26 pages. https://doi.org/10.1145/3656464

1 INTRODUCTION

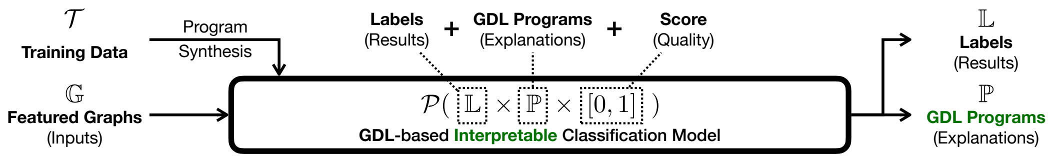
Learning on graphs has a wide variety of applications. Many significant real-world problems in diverse domains can be formulated as graph learning problems: healthcare [Zitnik et al. 2018], drug discovery [Li et al. 2022; Liu et al. 2022; Sun et al. 2019; Xiong et al. 2021], fraud detection [Rao et al. 2021], and program repair [Dinella et al. 2020]. In such decision-critical applications, users highly demand reliable explanations that elucidate the reasons for the classifications beyond

Authors' addresses: Minseok Jeon, Department of Computer Science and Engineering, Korea University, Republic of Korea, minseok_jeon@korea.ac.kr; Jihyeok Park, Department of Computer Science and Engineering, Korea University, Republic of Korea, jihyeok_park@korea.ac.kr; Hakjoo Oh, Department of Computer Science and Engineering, Korea University, Republic of Korea, hakjoo_oh@korea.ac.kr.

ACM 2475-1421/2024/6-ART234
https://doi.org/10.1145/3656464



(a) Existing approach: a *post-hoc* explanation of graph neural networks (GNNs) with *subgraphs*.



(b) Our approach: a *graph description language (GDL)*-based interpretable models.

Fig. 1. Two approaches for explainable graph learning.

accurate classification results. Therefore, developing an interpretable model or explainable artificial intelligence (XAI) [Gunning and Aha 2019] approach for graph learning problems is crucial.

Existing Approaches. Since graph neural networks (GNNs) are mainstream approaches to graph learning, most existing approaches to explainable graph learning have been focused on how to provide explanations of their classification results [Kakkad et al. 2023; Yuan et al. 2022]. GNNs are deep-learning-based models that have achieved remarkable performance in various graph learning problems. Despite their significant success, the predictions of GNNs are hardly explainable because of their deep and complex neural networks with millions of learning parameters optimized by training data. Thus, the current trend for explainable GNNs is to develop a separate post-hoc explanation model that provides explanations after their predictions. In a classification task, for example, the models produce explanations illustrating why the GNN made the classification results, as depicted in Figure 1a. While explanations can be defined in various forms, most existing techniques utilize subgraphs to explain the critical components of a given graph related to the classification results.

Limitations of Existing Approaches. However, the existing post-hoc explanation techniques with subgraphs for GNNs have two fundamental limitations: 1) **additional cost** and 2) **correctness** of explanations. First, they necessitate an additional cost to provide explanations after classification. Most existing techniques treat GNNs as black boxes and provide instance-wise explanations by searching for subgraphs highly correlated with the classification results in a given graph. Thus, this process is costly because the search space of subgraphs is exponential in the number of edges in a graph. Second, they cannot guarantee the correctness of explanations for the classification results. The explanation model separately learns the explanations from the original classification model and causes a semantic gap between the two models. It means that the explanations may not reflect the actual reasons why the classification model classifies a given graph into a specific label. These two limitations are challenging to be addressed as long as we rely on black-box models.

Our Approach. To address the limitations, we propose PL4XGL, a new programming language approach for inherently explainable graph learning for classification tasks. Figure 1b illustrates the overview of our approach. Our key idea is two-fold: 1) designing a **graph description language (GDL)** to explain the classification results and 2) developing a new **GDL-based interpretable classification model** instead of GNN-based models.

First, we design a graph description language, called GDL, as a declarative programming language in which a program describes a set of nodes, edges, or graphs. A program in GDL consists of node and edge descriptions and a target symbol. Node descriptions present symbolic nodes using variables and constraints on the variables, and edge descriptions present symbolic edges using pairs of variables and their constraints. Then, target symbol describes which components are targeted by the program. We utilize GDL programs as explanations for the classification results of graph learning problems (instead of subgraphs).

Then, we develop PL4XGL, a new GDL-based interpretable classification model that provides both classification results and explanations simultaneously. In our approach, a learned model consists of a set of candidate GDL programs with their labels and quality scores. In the classification process, PL4XGL searches for the best program that can describe the given graph data among its constituent GDL programs; PL4XGL classifies the graph data into the label that corresponds to the selected program. Simultaneously, the selected program is provided as an explanation for the classification (i.e., no additional explanation cost). Besides, the provided explanation (i.e., program) is guaranteed to be correct because PL4XGL made the classification based on the provided explanation.

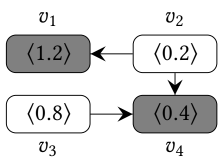
In our approach, learning a model is formulated as program synthesis [Alur et al. 2018; Gulwani et al. 2017] problem for candidate GDL programs with their labels and quality scores. To generate high-quality (e.g., precise and general) GDL programs, we adapt two synthesis algorithms, namely top-down [Feser et al. 2015; Frankle et al. 2016; Gulwani 2011] and bottom-up [Alur et al. 2017; Miltner et al. 2022; Udupa et al. 2013] methods, for our graph description language. Our top-down synthesis algorithm is designed to discover important feature values by exploring programs from simple to complex. Conversely, the bottom-up algorithm searches programs from complex to simple, and excels at capturing key graph structures.

We evaluate explainability of PL4XGL compared to the existing state-of-the-art GNN explanation technique SubgraphX [Yuan et al. 2021] in terms of 1) additional cost and 2) correctness of explanations. We use widely-used datasets [Park et al. 2022; Yuan et al. 2022] consisting of twelve synthetic and real-world datasets. Compared to GNN with SubgraphX, the classification & explanation step of PL4XGL is at least 35 times faster. To compare explainability, we utilize two widely used metrics, namely fidelity and sparsity [Yuan et al. 2022]. These serve as proxy metrics for measuring the correctness and conciseness of explanations, respectively. The experiment results show that PL4XGL always provides the optimal score for the fidelity metric (guaranteed by a theorem), while SubgraphX does not. Also, the experiment results on sparsity demonstrate that PL4XGL provides concise explanations. Finally, PL4XGL also accurately classifies graph data. Compared to six representative GNNs, PL4XGL shows competitive accuracy for the graph and node classification datasets. Especially, PL4XGL shows the best accuracy for three real-world molecular datasets, which are decision-critical datasets related to drug discovery.

Contributions. Our contributions are summarized as follows:

- We design a graph description language, called GDL, as a declarative programming language in which a program describes a set of nodes, edges, or graphs.
- We propose PL4XGL, a GDL-based interpretable classification model for explainable graph learning. We formulate learning our model as a program synthesis problem and present top-down and bottom-up synthesis algorithms.
- We experimentally demonstrate that PL4XGL is faster and provides more correct explanations compared to the existing state-of-the-art GNN explanation technique. PL4XGL also shows competitive classification accuracy compared to popular GNNs. Our implementation and datasets are publicly available[1].

[1]https://github.com/kupl/PL4XGL



(a) Featured graph G_1

$$\mathcal{M}_1 = \left\{ \begin{array}{c} (l_1, P_1, 0.9) \\ (l_2, P_2, 0.8) \\ (l_1, P_3, 0.0) \end{array} \right\}$$

(b) GDL-based model $\mathcal{M}_1$

$v_1 : (l_1, P_1, 0.9)$
$v_2 : (l_2, P_2, 0.8)$
$v_3 : (l_2, P_2, 0.8)$
$v_4 : (l_1, P_1, 0.9)$

(c) Classifications with explanations

```
node x <[0.0, 0.5]>
node y
edge (x, y)
target node y
```

(d) GDL program P_1

```
node x
node y <[0.2, 0.7]>
edge (x, y)
target node x
```

(e) GDL program P_2

```
node x
target node x
```

(f) GDL program P_3

Fig. 2. A running example: a GDL-based model $\mathcal{M}_1$ for node classification.

Scope. The focus of PL4XGL is on classification tasks. For example, regression tasks on graph data [Feng et al. 2023] are beyond the scope of this paper. Currently, our approach PL4XGL supports node, edge, and graph classification tasks.

2 INFORMAL OVERVIEW

In this section, we illustrate our approach using a simple node classification example.

Featured Graph. Figure 2a depicts a featured graph G_1 consisting of four nodes $\{v_1, v_2, v_3, v_4\}$ and three edges $\{(v_2, v_1), (v_2, v_4), (v_3, v_4)\}$. The gray-colored nodes (v_1 and v_4) belong to label l_1, while the white-colored nodes (v_2 and v_3) belong to label l_2. Each node is associated with a 1-dimensional feature vector; $\langle 1.2 \rangle$, $\langle 0.2 \rangle$, $\langle 0.8 \rangle$, and $\langle 0.4 \rangle$ for v_1, v_2, v_3, and v_4, respectively.

How Our Model Works. Figure 2b shows our GDL-based model $\mathcal{M}_1$ for node classification. It consists of three GDL programs: P_1, P_2, and P_3, described in Figures 2d, 2e, and 2f, respectively. The model $\mathcal{M}_1$ utilizes programs P_1 and P_3 to classify nodes into label l_1 and program P_2 for label l_2. The three GDL programs are scored based on their quality, where the scores indicate their precision in describing the nodes corresponding to the labels in the training data. The detailed definition of the score is described in Section 4. If a node is described by multiple GDL programs, the model classifies the node with the best scored program and provides the program as an explanation.

A GDL program describes which graph components (node, edge, or graphs) are classified into the corresponding label. It consists of variables, describing symbolic nodes with constraints, pairs of variables, describing symbolic edges, and a target symbol. In addition, we can interpret a GDL program in a natural language. For example, the GDL program P_1 describes:

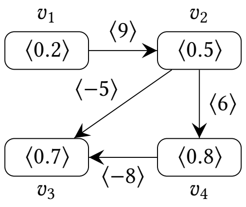
P_1: *"All nodes having a predecessor whose feature value is between 0.0 and 0.5"*

because 1) the target symbolic node is `y`, 2) the description of the symbolic edge `(x, y)` describes the existence of a predecessor `x`, and 3) the constraint of the variable `x` describes the possible feature values of the predecessor. Similarly, another GDL program P_2 describes:

P_2: *"All nodes having a successor whose feature value is between 0.2 and 0.7"*

The GDL program P_3 is the most general one, which describes all nodes in the graph.

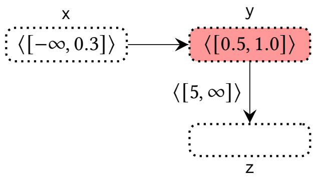
Our model accurately classifies all the nodes in the example graph and simultaneously provides correct explanations. Figure 2c shows the classification result with corresponding programs as explanations of why the model classifies each node into the label. The model classifies the nodes v_1 and v_4 into the label l_1 as P_1 is the best program describing them. Meanwhile, the nodes v_2 and v_3



(a) A featured graph G_2

```
node x <[    , 0.3]>
node y <[0.5, 1.0]>
node z
edge (x, y)
edge (y, z) <[5, ]>
target node y
```

(b) A GDL program P_4



(c) A graphical representation of P_4

Fig. 3. A running example of GDL

are classified into the label l_2 because of the program P_2. The used programs are simultaneously provided as explanations. The explanations are guaranteed to be correct because our model actually classified the nodes with the provided explanations. Existing GNNs, however, do not provide such correct explanations for their predictions; various GNN explanation techniques have been developed to explain their predictions. The produced explanations, however, are not guaranteed to be correct. That is, the provided explanations may not reflect the actual reason for the predictions.

Expressiveness of GDL. Intuitively, a GDL program is a set of subgraphs. For example, the first GDL program P_1 can be seen as a set of subgraphs describing node patterns as follows:

$$\{\boxed{\langle 1.2\rangle} \leftarrow \boxed{\langle 0.2\rangle},\ \boxed{\langle 0.4\rangle} \leftarrow \boxed{\langle 0.2\rangle},\ \boxed{\langle 0.0\rangle} \leftarrow \boxed{\langle 0.0\rangle},\ \boxed{\langle 0.0\rangle} \leftarrow \boxed{\langle 0.5\rangle},\ \ldots\}.$$

The first subgraph $\boxed{\langle 1.2\rangle} \leftarrow \boxed{\langle 0.2\rangle}$ describes a node pattern that the node and a predecessor have the feature values 1.2 and 0.2, respectively. The node v_1 in Figure 2a is described by the subgraph. The second subgraph $\boxed{\langle 0.4\rangle} \leftarrow \boxed{\langle 0.2\rangle}$, which describes v_4 in Figure 2a, illustrates a node pattern where the node and a predecessor have the feature values 0.4 and 0.2, respectively.

3 GRAPH DESCRIPTION LANGUAGE (GDL)

This section formally defines our *graph description language (GDL)*, which is a declarative programming language for describing target nodes, edges, or graphs themselves in featured graphs.

3.1 Featured Graphs

A *featured graph* $G = (V, E, \mathbf{F}_V, \mathbf{F}_E) \in \mathbb{G}$ is a graph defined with feature vectors for nodes and edges:

- $V = \{v_1, v_2, \ldots, v_n\}$ is a set of n **nodes**.
- $E = \{e_1, e_2, \ldots, e_m\} \subseteq V \times V$ is a set of m **edges**.
- $\mathbf{F}_V \subseteq \mathbb{R}^{n\times d}$ is a **node feature matrix** for d-dimensional node feature vectors.
- $\mathbf{F}_E \subseteq \mathbb{R}^{m\times c}$ is an **edge feature matrix** for c-dimensional edge feature vectors.

The i-th row of $\mathbf{F}_V$ or $\mathbf{F}_E$ corresponds to the feature vector of the i-th node v_i or edge e_i, respectively. We use $\mathbf{f}^G_v$ and $\mathbf{f}^G_{(v,v')}$ to denote the feature vector of a node $v \in V$ and an edge $(v, v') \in E$, respectively, in a featured graph G.

Example. Figure 3a depicts a featured graph G_2 with 1-dimensional node and edge features:

$$G_2 = \begin{pmatrix} V = \{ & v_1, & v_2, & v_3, & v_4\}, & E = \{ & (v_1, v_2), & (v_2, v_3), & (v_2, v_4), & (v_4, v_3)\}, \\ \mathbf{F}_V = \langle & \langle 0.2\rangle, & \langle 0.5\rangle, & \langle 0.7\rangle, & \langle 0.8\rangle\rangle, & \mathbf{F}_E = \langle & \langle 9\rangle, & \langle -5\rangle, & \langle 6\rangle, & \langle -8\rangle\rangle \end{pmatrix}$$

In this example, $\mathbf{f}^{G_2}_{v_2} = \langle 0.5\rangle$ denotes the feature vector of the node v_2 in the featured graph G_2, and $\mathbf{f}^{G_2}_{(v_2,v_4)} = \langle 6\rangle$ denotes the feature vector of the edge (v_2, v_4) in the featured graph G_2.

$$
\begin{array}{llcll}
\text{Programs} & P & ::= \overline{\delta}\ \texttt{target}\ t & \in & \mathbb{P} = \mathbb{D}^* \times \mathbb{T} \\
\text{Descriptions} & \delta & ::= \delta_V \mid \delta_E & \in & \mathbb{D} = \mathbb{D}_V \uplus \mathbb{D}_E \\
\text{Node Descriptions} & \delta_V & ::= \texttt{node}\ x \,\texttt{<}\overline{\phi}\texttt{>}^? & \in & \mathbb{D}_V = \mathbb{X} \times \Phi^d \\
\text{Edge Descriptions} & \delta_E & ::= \texttt{edge}\ (x\texttt{,}x)\ \texttt{<}\overline{\phi}\texttt{>}^? & \in & \mathbb{D}_E = \mathbb{X} \times \mathbb{X} \times \Phi^c \\
\text{Target Symbols} & t & ::= \texttt{node}\ x \mid \texttt{edge}\ (x\texttt{,}x) \mid \texttt{graph} & \in & \mathbb{T} = \mathbb{X} \uplus (\mathbb{X} \times \mathbb{X}) \uplus \{\epsilon\} \\
\text{Intervals} & \phi & ::= \texttt{[}n^?\texttt{,}n^?\texttt{]} & \in & \Phi = (\mathbb{R} \uplus \{-\infty\}) \times (\mathbb{R} \uplus \{\infty\}) \\
\text{Real Numbers} & n & ::= \texttt{0.2} \mid \texttt{0.7} \mid \texttt{6} \mid \texttt{-8} \ \ldots & \in & \mathbb{R} \\
\text{Variables} & x & ::= \texttt{x} \mid \texttt{y} \mid \texttt{z} \mid \ldots & \in & \mathbb{X}
\end{array}
$$

Fig. 4. The syntax of GDL.

$$
\begin{array}{lll}
[\![\texttt{<}\phi_1\texttt{,}\ldots\texttt{,}\phi_k\texttt{>}]\!] & : \mathcal{P}(\mathbb{R}^k) & = \{\ \mathbf{f} \mid \mathbf{f} = \langle f_1, \ldots, f_k \rangle \wedge \forall i.\ \phi_i = [a, b] \Rightarrow a \le f_i \le b\} \\[2ex]
[\![\texttt{node}\ x \,\texttt{<}\overline{\phi}\texttt{>}]\!] & : \mathcal{P}(\mathbb{G} \times \mathbb{H}) & = \{\ (G, \eta) \mid v = \eta(x) \wedge \mathbf{f}^G_v \in [\![\texttt{<}\overline{\phi}\texttt{>}]\!]\} \\
[\![\texttt{edge}\ (x\texttt{,}y)\ \texttt{<}\overline{\phi}\texttt{>}]\!] & : \mathcal{P}(\mathbb{G} \times \mathbb{H}) & = \{\ (G, \eta) \mid e \in E \wedge e = (\eta(x), \eta(y)) \wedge \mathbf{f}^G_e \in [\![\texttt{<}\overline{\phi}\texttt{>}]\!]\} \\
[\![\delta_1 \delta_2 \ldots \delta_k]\!] & : \mathcal{P}(\mathbb{G} \times \mathbb{H}) & = \{\ (G, \eta) \mid \forall i.\ (G, \eta) \in [\![\delta_i]\!]\} \\[2ex]
[\![\overline{\delta}\ \texttt{target}\ \texttt{node}\ x]\!] & : \mathcal{P}(V) & = \{\ v \mid \exists (G, \eta) \in [\![\overline{\delta}]\!].\ v = \eta(x)\} \\
[\![\overline{\delta}\ \texttt{target}\ \texttt{edge}\ (x, y)]\!] & : \mathcal{P}(E) & = \{\ e \mid \exists (G, \eta) \in [\![\overline{\delta}]\!].\ e = (\eta(x), \eta(y))\} \\
[\![\overline{\delta}\ \texttt{target}\ \texttt{graph}]\!] & : \mathcal{P}(\mathbb{G}) & = \{\ G \mid \exists (G, \eta) \in [\![\overline{\delta}]\!]\}
\end{array}
$$

Fig. 5. The semantics of GDL where $G = (V, E, \mathbf{F}_V, \mathbf{F}_E) \in \mathbb{G}$ is a given featured graph.

3.2 Syntax of GDL

Figure 4 formally defines the syntax of GDL. We use the notation $\overline{A}$ to denote a sequence of elements in A, and the notation $A^?$ to denote an optional element in A. A GDL *program* $P \in \mathbb{P}$ consists of a sequence of descriptions $\overline{\delta} \in \mathbb{D}$ and a target symbol $t \in \mathbb{T}$. A *description* δ is either a node description $\delta_V \in \mathbb{D}_V$ or an edge description $\delta_E \in \mathbb{D}_E$. A *node description* **node** $x\,\texttt{<}\phi_1\texttt{,}\ldots\texttt{,}\phi_d\texttt{>}$ introduces a new variable $x \in \mathbb{X}$ for a symbolic node whose d-dimensional feature vector is bounded by the interval vector $\texttt{<}\phi_1, \ldots, \phi_d\texttt{>}$. An *edge description* **edge** $(x, y)\ \texttt{<}\phi_1\texttt{,}\ldots\texttt{,}\phi_c\texttt{>}$ describes a symbolic edge from a symbolic node x to a symbolic node y whose c-dimensional feature vector is bounded by the interval vector $\texttt{<}\phi_1, \ldots, \phi_c\texttt{>}$. If no bound is specified in an interval, the default lower (or upper) bound is $-\infty$ (or ∞). Finally, a *target symbol* $t \in \mathbb{T}$ is either 1) a symbolic node **node** x, 2) a symbolic edge **edge** $(x\texttt{,}y)$, or 3) a graph **graph** itself.

Example. Figure 3b shows a GDL program P_4 that describes nodes by target symbol **node** y in featured graphs with 1-dimensional node and edge features. The program introduces three symbolic nodes with variables x, y, and z with the interval vectors $\langle[-\infty, 0.3]\rangle$, $\langle[0.5, 1.0]\rangle$, and $\langle[-\infty, \infty]\rangle$, respectively. Similarly, it contains two edge descriptions for symbolic edges (x, y) and (y, z), and the corresponding edge features are interval vectors $\langle[-\infty, \infty]\rangle$ and $\langle[5, \infty]\rangle$, respectively. Finally, the targets of the program are nodes described by the variable y. Figure 3c shows a graphical representation of the program P_4. Each node and edge in the graph represents a node and edge description in the program, respectively, and the target symbolic node is highlighted in red. For brevity, we omit the default interval vector $\langle[-\infty, \infty]\rangle$ in the graphical representation.

3.3 Semantics of GDL

Now, we define the semantics of GDL. We first define the semantics of interval vectors and node/edge descriptions using valuations of symbolic nodes. Then, we define the semantics of GDL programs.

Interval Vectors. For a k-dimensional interval vector $<\overline{\phi}> = <\phi_1, \ldots, \phi_k> \in \Phi^k$, its semantics $[\![<\overline{\phi}>]\!] : \mathcal{P}(\mathbb{R}^k)$ is defined as a set of k-dimensional feature vectors where each dimension is bounded by the corresponding interval.

Descriptions. The semantics $[\![\delta]\!] : \mathcal{P}(\mathbb{G} \times \mathbb{H})$ of a description $\delta \in \mathbb{D}$ is defined as a set of pairs of a featured graphs and valuations that satisfy the description δ. A *valuation* $\eta \in \mathbb{H} = \mathbb{X} \to V$ is a mapping from each variable denoting symbolic nodes in GDL program into a distinct concrete node in the featured graph (i.e., $x, x' \in \mathbb{X}. x \neq x' \implies \eta(x) \neq \eta(x')$). In other words, a valuation η assigns a distinct concrete node $\eta(x) \in V$ to each variable $x \in \mathbb{X}$, and represents a subgraph $G|_\eta$ of G consisting of the following nodes and edges:

- $V|_\eta = \{\eta(x) \mid x \in \mathbb{X}\}$, and
- $E|_\eta = \{(\eta(x), \eta(y)) \mid (\eta(x), \eta(y)) \in E \wedge (x, y) \in \mathbb{X} \times \mathbb{X}\}$.

Among possible valuations, the description δ allows only a pair (G, η) of a featured graph and a valuation that satisfies its interval vector. The pair (G, η) satisfies a node description $\delta_V = \textbf{node}\ x <\overline{\phi}>$ if and only if the feature vector of the concrete node $\eta(x)$ is bounded by the interval vector $<\overline{\phi}>$. Similarly, it satisfies an edge description $\delta_E = \textbf{edge}\ (x, y) <\overline{\phi}>$ if and only if the feature vector of the concrete edge $(\eta(x), \eta(y))$ exists in E and is bounded by the interval vector $<\overline{\phi}>$. The semantics $[\![\overline{\delta}]\!] : \mathcal{P}(\mathbb{G} \times \mathbb{H})$ of a sequence of node/edge descriptions $\overline{\delta} \in \mathbb{D}^*$ is defined as a set of pairs of featured graph and valuation that satisfy all descriptions in $\overline{\delta}$.

GDL Programs. The type of semantics $[\![P]\!]$ of a GDL program P varies depending on its targets: 1) nodes, 2) edges, and 3) graphs. First, a program defined with a target variable describes a set of featured graphs with nodes. Its semantics $[\![\overline{\delta}\ \textbf{target}\ \textbf{node}\ x]\!] : \mathcal{P}(V)$ collects nodes that can be assigned to the target symbolic node x by any valuation η satisfying all the descriptions $\overline{\delta}$. Similarly, $[\![\overline{\delta}\ \textbf{target}\ \textbf{edge}\ (x, y)]\!] : \mathcal{P}(E)$ collects edges that can be assigned to the target symbolic edge (x, y) by any valuation η satisfying all the descriptions $\overline{\delta}$. If the target symbol is **graph**, the program semantics $[\![\overline{\delta}\ \textbf{target}\ \textbf{graph}]\!] : \mathcal{P}(\mathbb{G})$ describes a set of graphs containing at least one subgraph $G|_\eta$ by a valuation η satisfying all the descriptions $\overline{\delta}$.

Example. Figure 3a depicts a featured graph G_2, and Figure 3b shows a GDL program P_4:

$$P_4 = \begin{tikzcd} \overset{\mathrm{x}}{\boxed{\langle[-\infty, 0.3]\rangle}} \arrow[r] & \overset{\mathrm{y}}{\boxed{\langle[0.5, 1.0]\rangle}} \arrow[r, "{\langle[5, \infty]\rangle}"] & \overset{\mathrm{z}}{\boxed{}} \end{tikzcd}.$$

Now, consider the following valuation $\eta : \mathbb{X} \to V$:

$$\eta = \{ \quad \mathrm{x} \to v_1, \quad \mathrm{y} \to v_2, \quad \mathrm{z} \to v_4 \quad \}.$$

Then, it represents the following subgraph $G_2|_\eta$ of the featured graph G_2:

$$G_2|_\eta = \begin{tikzcd} \overset{\eta(\mathrm{x}) = v_1}{\boxed{\langle 0.2\rangle}} \arrow[r, "\langle 9\rangle"] & \overset{\eta(\mathrm{y}) = v_2}{\boxed{\langle 0.5\rangle}} \arrow[r, "\langle 6\rangle"] & \overset{\eta(\mathrm{z}) = v_4}{\boxed{\langle 0.8\rangle}} \end{tikzcd}$$

and satisfies all the node/edge descriptions $\overline{\delta}$ in P_4 (i.e., $(G_2, \eta) \in [\![\overline{\delta}]\!]$). For example, the valuation η satisfies the node description **node** `y <[0.5, 1.0]>` because the concrete node assigned by η for `y` is v_2 and $\mathbf{f}_{v_2}^{G_2} = \langle 0.5\rangle$ is bounded by the interval vector $\langle[0.5, 1.0]\rangle$. It also satisfies the edge description **edge** `(y, z) <[5, ]>` because the concrete nodes assigned by η for `y` and `z` are v_2 and v_4, respectively, and $\mathbf{f}_{(v_2, v_4)}^{G_2} = \langle 6\rangle$ is bounded by the interval vector $\langle[5, \infty]\rangle$. Since the target symbol of the program P_4 is a symbolic node **node** `y`, the node $\eta(\mathrm{y}) = v_2$ is in its semantics $[\![P_4]\!]$ (i.e., $v_2 \in [\![P_4]\!]$). That is, the program P_4 describes the node v_2 under the valuation η.

3.4 Generality Order between GDL Programs

We define a partial order ($\sqsubseteq$) between GDL programs in terms of the generality of their semantics:

$$P \sqsubseteq P' \iff [\![P]\!] \subseteq [\![P']\!].$$

Intuitively, $P \sqsubseteq P'$ means that P' is *more general* than P because P' can describe all the graph P can describe. In addition, we use the notation $P \sqsubset P'$ to denote that P' is *strictly more general* than P:

$$P \sqsubset P' \iff (P \sqsubseteq P') \land ([\![P]\!] \neq [\![P']\!]).$$

Several mutation operations on GDL programs might increase or decrease the generality of GDL programs. For example, $P_5 = \overset{\text{x}}{\langle[-\infty, 0.3]\rangle} \to \overset{\text{y}}{\langle[0.5, 1.0]\rangle}$ is a program mutated from P_4 in Figure 3b by removing the variable z and its related descriptions. Then, P_5 is strictly more general than P_4 (i.e., $P_4 \sqsubset P_5$). Our program synthesis algorithm (Section 5) utilizes the order to search high-quality GDL programs.

4 PL4XGL: A GDL-BASED EXPLAINABLE CLASSIFICATION MODEL

Now, we define our GDL-based explainable classification model PL4XGL.

4.1 Classification Tasks on Featured Graphs

A *classification task* $\mathbb{C} \to \mathbb{L}$ on featured graphs is defined as a problem of classifying each graph component $c \in \mathbb{C}$ in featured graph to a label $l \in \mathbb{L}$. We consider three types of classification tasks on featured graphs with different graph components: 1) *nodes*, 2) *edges*, and 3) *graphs*:

- **Node Classification**: $(\mathbb{G} \times V) \to \mathbb{L}$
- **Edge Classification**: $(\mathbb{G} \times E) \to \mathbb{L}$
- **Graph Classification**: $\mathbb{G} \to \mathbb{L}$

A node or edge classification task is defined as a problem of classifying each node $v \in V$ or each edge $e \in E$ of a given featured graph $G \in \mathbb{G}$ to a label $l \in \mathbb{L}$, respectively. A graph classification task is defined as a problem of classifying a given featured graph $G \in \mathbb{G}$ itself to a label $l \in \mathbb{L}$.

4.2 Explainable Classification Model

A *GDL-based explainable classification model* $\mathcal{M} \in \mathbb{M}$ is defined as a set of candidate GDL programs with their labels and quality scores:

$$\mathcal{M} \in \mathbb{M} = \mathcal{P}(\mathbb{L} \times \mathbb{P} \times [0, 1])$$

where each candidate $(l, P, \psi) \in \mathcal{M}$ consists of:

- a label $l \in \mathbb{L}$ as the **classification result**,
- a GDL program $P \in \mathbb{P}$ as the **explanation** of the result, and
- a real number $\psi \in [0, 1]$ as the **quality score** of the pair (l, P).

The quality score ψ reflects the *precision* of the program P for the classification result l, and we will explain how to compute it in Section 5. For a classification task $\mathbb{C} \to \mathbb{L}$ on featured graphs, the *Search* : $(\mathbb{M} \times \mathbb{C}) \to (\mathbb{L} \times \mathbb{P} \times [0, 1])$ function takes a model $\mathcal{M}$ and returns the best candidate $(l, P, \psi) \in \mathcal{M}$ that can describe the given graph component $c \in \mathbb{C}$:

$$\mathit{Search}(\mathcal{M}, c) = \operatorname*{argmax}_{(l,P,\psi)\in\mathcal{M}} \begin{cases} \psi & \text{if } c \in [\![P]\!] \\ -1 & \text{otherwise} \end{cases}.$$

In other words, it returns not only the classification result l but also the selected GDL program P as its explanation and the quality score ψ of the pair of classification and explanation. To handle the case when there is no possible candidate GDL program describing c, we make a model $\mathcal{M}$ contain the most general GDL program $P_\top$ [2] with a label l' and the lowest quality score (i.e., $\psi = 0.0$) (i.e., $(P_\top, l', 0.0) \in \mathcal{M}$). Then, any component c can be described by at least one program in the model.

5 LEARNING GDL-BASED MODEL USING PROGRAM SYNTHESIS

Now, we present our learning algorithm. The core idea of our learning algorithm for GDL-based models is to iteratively synthesize better candidate GDL programs from the training data.

Training Data. A set of *training data* $\mathcal{T} \in \mathbb{T} = \mathcal{P}(\mathbb{C} \times \mathbb{L})$ is a set of pairs of a graph component $c \in \mathbb{C}$ and a label $l \in \mathbb{L}$. It means that the expected classification result of the graph component c is the label l. We utilize the notation $\mathcal{T}(l) = \{c \mid (c, l) \in \mathcal{T}\}$ to denote the set of graph components in the training data $\mathcal{T}$ that are expected to be classified into the label l.

Quality Score. A *quality score* $\psi \in [0, 1]$ is a real number between 0 and 1 that reflects the *precision* of a candidate GDL program P for the classification result l according to the training data $\mathcal{T}$. The *Score* : $\mathbb{T} \to \mathbb{L} \times \mathbb{P} \to [0, 1]$ function computes the quality score ψ:

$$\psi = Score(\mathcal{T})(l, P) = \frac{|\llbracket P \rrbracket \cap \mathcal{T}(l)|}{|\llbracket P \rrbracket| + \epsilon}$$

where a hyperparameter ϵ is a small positive real number and added to avoid the overfitting problem or handle noisy data. While the quality score ψ computed by the *Score* function basically reflects the precision for each candidate in the model $\mathcal{M}$, it becomes higher when the precision is computed by a larger number of graph components. We chose the hyperparameter ϵ in $\{0.1, 1.0, 10.0\}$ that maximized the classification accuracy on the validation set in our evaluation.

Learning Objective. The *learning objective* of our algorithm is to find a GDL-based model $\mathcal{M}$ that satisfies the following three properties using program synthesis over the training data $\mathcal{T}$:

(1) **Coverage**: at least one non-trivial (i.e., not the most general) candidate GDL program P in the model $\mathcal{M}$ *covers* each graph component c in the training data $\mathcal{T}$:

$$\forall l \in \mathbb{L}.\ \forall c \in \mathcal{T}(l).\ \exists (l, P, \_) \in \mathcal{M}.\ (P \neq P_\top) \wedge (c \in \llbracket P \rrbracket).$$

(2) **Precision**: the model $\mathcal{M}$ maximizes the average quality score ψ that reflects the *precision* of each pair of a candidate GDL and a label in the model $\mathcal{M}$:

$$\frac{\sum\{\psi \mid (l, P, \psi) \in \mathcal{M}\}}{|\mathcal{M}|}.$$

(3) **Generality**: the model $\mathcal{M}$ consists of as much general candidate programs as possible; $P_1 \sqsubset P_2$ means P_2 is *strictly more general* than P_1, as defined in Section 3.4:

$$P \sqsubset P' \iff (P \sqsubseteq P') \wedge (\llbracket P \rrbracket \neq \llbracket P' \rrbracket).$$

5.1 Algorithm Outline

Algorithm 1 presents our learning algorithm. It takes a set of training data $\mathcal{T} : \mathcal{P}(\mathbb{L} \times \mathbb{C})$ as input and returns learned GDL-based model $\mathcal{M}$. The Learn procedure at lines 1–7 describes the overall learning algorithm. First, it initializes the model $\mathcal{M}$ as an empty set (line 2). At lines 3–6, the procedure synthesizes a GDL program from each $(c, l) \in \mathcal{T}$. After synthesizing a program P for the graph component c and the label l (line 4), it computes its quality score ψ (line 5) and adds the candidate (l, P, ψ) to the model $\mathcal{M}$ (line 6). Finally, if no more training data exists, the procedure returns the learned model $\mathcal{M}$ (line 7). Note that the iterations at lines 3–6 are independent of each other; each GDL program can be synthesized *in parallel*.

[2] $P_\top$ = `node x target node x` for node classification, $P_\top$ = `node x node y edge (x,y) target edge (x, y)` for edge classification, and $P_\top$ = `target graph` for graph classification.

Algorithm 1 Learning algorithm

```
Require: A set of training data $\mathcal{T} : \mathcal{P}(\mathbb{L} \times \mathbb{C})$
Ensure: A GDL-based model $\mathcal{M} \in \mathbb{M}$
1:  procedure LEARN($\mathcal{T}$)
2:      $\mathcal{M} \leftarrow \varnothing$
3:      for each $(c, l) \in \mathcal{T}$ do
4:          $P \leftarrow$ SYNTHESIZE($\mathcal{T}, c, l$)
5:          $\psi \leftarrow Score(\mathcal{T})(l, P)$
6:          $\mathcal{M} \leftarrow \mathcal{M} \cup \{(l, P, \psi)\}$
7:      return $\mathcal{M}$
8:  procedure SYNTHESIZE($\mathcal{T}, c, l$)
9:      $P \leftarrow Initialize(c)$
10:     $updated \leftarrow true$
11:     while $updated$ do
12:         $\overline{P}_1 \leftarrow Mutate(P)$
13:         $\overline{P}_2 \leftarrow FilterBetter(\mathcal{T}, c, l)(P, \overline{P}_1)$
14:         if $\overline{P}_2 = \varnothing$ then $updated \leftarrow false$
15:         else $P \leftarrow Select(\overline{P}_2)$
16:     return $P$
```

The SYNTHESIZE procedure at lines 8–16 presents a generic template for the synthesis algorithm, and the *Initialize* : $\mathbb{C} \rightarrow \mathbb{P}$ and *Mutate* : $\mathbb{P} \rightarrow \mathcal{P}(\mathbb{P})$ functions are instantiated differently in the top-down and bottom-up algorithms. It first initializes the candidate GDL program P using the *Initialize* function with a given graph component c (line 9) and the *updated* flag as *true* (line 10). At lines 11–15, it iteratively updates the candidate GDL program P until no more updates are possible. The procedure first mutates P into a set of all possible mutated programs $\overline{P}_1$ using the *Mutate* function (line 12). Then, it collects only the better programs $\overline{P}_2$ in $\overline{P}_1$ than the current best program P using the *FilterBetter* : $(\mathbb{T} \times \mathbb{C} \times \mathbb{L}) \rightarrow (\mathbb{P} \times \mathcal{P}(\mathbb{P})) \rightarrow \mathcal{P}(\mathbb{P})$ function (line 13):

$$FilterBetter(\mathcal{T}, c, l)(P, \overline{P}_1) = \{P' \in \overline{P}_1 \mid (\text{①}\ c \in [\![P']\!]) \wedge (\text{②}\ (\psi < \psi') \vee (\text{③}\ \psi = \psi' \wedge P \sqsubset P'))\}$$

where $\psi = Score(\mathcal{T})(l, P)$ and $\psi' = Score(\mathcal{T})(l, P')$. It consider the learning objective of the model: 1) *coverage*, 2) *precision*, and 3) *generality* in order. First, for *coverage* of the model, the function first filters out the programs that cannot describe the graph given component c to ensure that c is always covered by the model. Second, to maximize the *precision* of the model, the function only considers the programs that have better quality scores than the current best program P. Then, it checks the new program P' is strictly more general than the current one P for better *generality* of the model. If no programs are better than P in (line 14), the iteration terminates. Otherwise, it picks any program in $\overline{P}_2$ using *Select* : $\mathcal{P}(\mathbb{P}) \rightarrow \mathbb{P}$ function and updates P (line 15). Finally, the procedure returns the current best program P (line 16).

The high-level idea of the top-down (or bottom-up) algorithm is to iteratively specialize (or generalize) descriptions in the program from the most general (or most specific) one. In our evaluation, we choose a better algorithm that performs better for the validation sets: the top-down algorithm for the node classification task and the bottom-up algorithm for the graph classification task. Now, we describe the details of the top-down and bottom-up algorithms.

5.2 Top-Down Algorithm

The *top-down* algorithm starts from the most general program and iteratively mutates it into a more specific one to increase the precision of the program while preserving the coverage of the program. For the top-down algorithm, the *Initialize* : $\mathbb{C} \rightarrow \mathbb{P}$ function returns the *most general* GDL program $P_\top$ for whatever graph component c is given. The *Mutate* : $\mathbb{P} \rightarrow \mathcal{P}(\mathbb{P})$ function returns the set of mutated programs:

$$Mutate(P) = \{P' \mid P \overset{\downarrow}{\rightsquigarrow} P'\}$$

where $\overset{\downarrow}{\rightsquigarrow}$ denotes one-step mutation of the program for the top-down algorithm in Figure 6.

Each one-step mutation $P \overset{\downarrow}{\rightsquigarrow} P'$ is an operation that *strictly decreases* the generality of the program (i.e., $P' \sqsubset P$) by adding descriptions (*AddNode* or *AddEdge*) or specializeing features (*SpecializeNode* or *SpecializeEdge*). The *AddNode* rule adds a new node description $\delta_V = (x, <\overline{\phi}>)$

$$\frac{\begin{array}{cc}(x,\_) \notin \bar{\delta} & \delta_V = (x,<\bar{\phi}>) \\ (y,\_) \in \bar{\delta} & \delta_E = (x,y,<\bar{\phi}'>) \text{ or } (y,x,<\bar{\phi}'>)\end{array}}{(\bar{\delta}, t) \overset{\downarrow}{\rightsquigarrow} (\bar{\delta} \cup \{\delta_V, \delta_E\}, t)}\ \textit{AddNode} \qquad \frac{\begin{array}{cc}(x,\_) \in \bar{\delta} & (x,y,\_) \notin \bar{\delta} \\ (y,\_) \in \bar{\delta} & \delta_E = (x,y,<\bar{\phi}>)\end{array}}{(\bar{\delta}, t) \overset{\downarrow}{\rightsquigarrow} (\bar{\delta} \cup \{\delta_E\}, t)}\ \textit{AddEdge}$$

$$\frac{\begin{array}{c}\delta_V = (x,<\bar{\phi}>) \in \bar{\delta} \\ <\bar{\phi}'> \in \textit{SpecializeItv}(<\bar{\phi}>) \\ \delta_V' = (x,<\bar{\phi}'>)\end{array}}{(\bar{\delta}, t) \overset{\downarrow}{\rightsquigarrow} (\bar{\delta} \setminus \{\delta_V\} \cup \{\delta_V'\}, t)}\ \textit{SpecializeNode} \qquad \frac{\begin{array}{c}\delta_E = (x,y,<\bar{\phi}>) \in \bar{\delta} \\ <\bar{\phi}'> \in \textit{SpecializeItv}(<\bar{\phi}>) \\ \delta_E' = (x,y,<\bar{\phi}'>)\end{array}}{(\bar{\delta}, t) \overset{\downarrow}{\rightsquigarrow} (\bar{\delta} \setminus \{\delta_E\} \cup \{\delta_E'\}, t)}\ \textit{SpecializeEdge}$$

Fig. 6. One-step mutation rules ($\overset{\downarrow}{\rightsquigarrow}$) for top-down algorithm.

and an edge description δ_E from the node x to a existing node y or vice versa into the program P. The *AddEdge* rule adds a new edge description $\delta_E = (x, y, <\bar{\phi}>)$ between existing variables x and y into the program P. The *SpecializeNode* (or *SpecializeEdge*) rule specialize the interval vector of the existing symbolic node x (or symbolic edge (x, y)) in the program P by using *SpecializeItv*.

The implementation of *SpecializeItv* (i.e., specializing the intervals) is a design choice. Our *SpecializeItv* cuts an interval with the median value in the interval. Due to the space limit, we present our implementation of *SpecializeItv* in Section A of our supplementary material. The mutation function (i.e., *Mutate*) is also a design choice. For instance, the following mutation function $\textit{Mutate}^2$ can be used which enumerates a larger number of mutated programs:

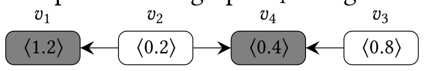
$$\textit{Mutate}^2(P) = \textit{Mutate}(P) \cup \bigcup_{P' \in \textit{Mutate}(P)} \textit{Mutate}(P').$$

Similarly, we can define $\textit{Mutate}^k$, where k determines the enumeration depth, as follows:

$$\textit{Mutate}^k(P) = \textit{Mutate}^{k-1}(P) \cup \bigcup_{P' \in \textit{Mutate}^{k-1}(P)} \textit{Mutate}(P').$$

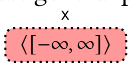
Using $\textit{Mutate}^k$ ($k > 1$) instead of *Mutate* may find better-scored GDL programs, but this exponentially increases the training cost. In our evaluation, we used $\textit{Mutate}^3$ for the two small synthetic datasets, while we used *Mutate* for the other real-world datasets. The impact of the depth k is discussed at the end of Section 6.3.

Example. Recall the following example featured graph G_1 in Figure 2a

$G_1 =$ 

where the gray and white nodes belong to label l_1 and l_2, respectively. Suppose the current target training component of the Synthesize procedure is the node v_1, and the hyperparameter ϵ is 1. We ignore edge features in the graph and edge descriptions in GDL programs for brevity. The top-down algorithm explores program space $\mathbb{P}$ from the most general one to a more specific one.

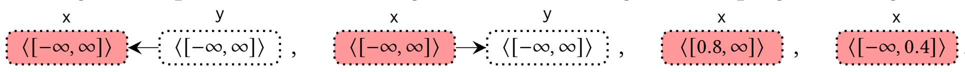
(1) The algorithm first generates the most general program for node classification:



.

Then, it describes all nodes $\{v_1, v_2, v_3, v_4\}$, and its quality score ψ is $\frac{|\{v_1,v_2\}|}{|\{v_1,v_2,v_3,v_4\}|+1} = \frac{2}{5}$.

(2) Using one-step mutation rules in Figure 6, the following mutated programs are generated:



The first two programs are generated by the *AddNode* rule, and the last two are generated by the *SpecializeNode* rule. The last two programs specified the interval with 0.4 and 0.8, which

$$\frac{\begin{array}{l}\delta_V = (x, \_) \in \overline{\delta} \\ D_x = \{\delta_E \in \overline{\delta} \mid \delta_E = (x, \_, \_) \lor \delta_E = (\_, x, \_)\}\end{array}}{(\overline{\delta}, t) \overset{\uparrow}{\rightsquigarrow} (\overline{\delta} \setminus \{\delta_V\} \setminus D_x, t)}\ \textit{RemoveNode} \qquad \frac{\delta_E \in \overline{\delta}}{(\overline{\delta}, t) \overset{\uparrow}{\rightsquigarrow} (\overline{\delta} \setminus \{\delta_E\}, t)}\ \textit{RemoveEdge}$$

$$\frac{\begin{array}{l}\delta_V = (x, <\overline{\phi}>) \in \overline{\delta} \\ <\overline{\phi}'> \in \textit{GeneralizeItv}(<\overline{\phi}>) \\ \delta'_V = (x, <\overline{\phi}'>)\end{array}}{(\overline{\delta}, t) \overset{\uparrow}{\rightsquigarrow} (\overline{\delta} \setminus \{\delta_V\} \cup \{\delta'_V\}, t)}\ \textit{GeneralizeNode} \qquad \frac{\begin{array}{l}\delta_E = (x, y, <\overline{\phi}>) \in \overline{\delta} \\ <\overline{\phi}'> \in \textit{GeneralizeItv}(<\overline{\phi}>) \\ \delta'_E = (x, y, <\overline{\phi}'>)\end{array}}{(\overline{\delta}, t) \overset{\uparrow}{\rightsquigarrow} (\overline{\delta} \setminus \{\delta_E\} \cup \{\delta'_E\}, t)}\ \textit{GeneralizeEdge}$$

Fig. 7. One-step mutation rules ($\overset{\uparrow}{\rightsquigarrow}$) for bottom-up algorithm.

are median feature values of the training nodes. Among them, the first program is selected with the score $\psi = \frac{|\{v_1, v_4\}|}{|\{v_1, v_4\}|+1} = \frac{2}{3}$, which precisely includes the nodes belonging to l_1.

(3) Now, since no more better program exists as the result of the mutation, the algorithm terminates and returns the program:

$$\begin{tikzcd} \overset{\textsf{x}}{\boxed{\langle[-\infty, \infty]\rangle}} & \overset{\textsf{y}}{\boxed{\langle[-\infty, \infty]\rangle}} \arrow[l] \end{tikzcd}.$$

5.3 Bottom-Up Algorithm

The *bottom-up* algorithm works oppositely; it starts from the most specific program and iteratively mutates it into a more general one while preserving the precision of the program. For the bottom-up algorithm, the *Initialize* : $\mathbb{C} \longrightarrow \mathbb{P}$ function returns the *most specific* GDL program P that describes the given graph component c, satisfying $\forall P' \sqsubset P. c \notin [\![P']\!]$. The implementation of *Initialize* is decribed in Section A of our supplementary material. The *Mutate* : $\mathbb{P} \longrightarrow \mathcal{P}(\mathbb{P})$ function returns the set of mutated programs:

$$\textit{Mutate}(P) = \{P' \mid P \overset{\uparrow}{\rightsquigarrow} P'\}$$

where $\overset{\uparrow}{\rightsquigarrow}$ denotes one-step mutation of the program for the bottom-up algorithm in Figure 7, and it *strictly increases* the generality of the program P. We skip the detailed description of the $\overset{\uparrow}{\rightsquigarrow}$ rules because they are similar to the $\overset{\downarrow}{\rightsquigarrow}$ rules in Figure 6 but with opposite semantics. How to generalize interval vectors for program mutation (i.e., *GeneralizeItv*) is also a design choice. We generalize an interval by replacing a lower bound or an upper bound with $-\infty$ or ∞, respectively. The detailed implementation of our *GeneralizeItv* is described in Section A of our supplementary material.

Example. Consider the example in Section 5.2 again but now with the bottom-up algorithm.

(1) The algorithm first generates the most specific program for the node v_1 in G_1:

$$\begin{tikzcd} \overset{\textsf{a}}{\boxed{\langle[1.2, 1.2]\rangle}} & \overset{\textsf{b}}{\boxed{\langle[0.2, 0.2]\rangle}} \arrow[l] \arrow[r] & \overset{\textsf{c}}{\boxed{\langle[0.4, 0.4]\rangle}} \arrow[r] & \overset{\textsf{d}}{\boxed{\langle[0.8, 0.8]\rangle}} \end{tikzcd}.$$

Then, it can describe only v_1, and its quality score is $\psi = \frac{|\{v_1\}|}{|\{v_1\}|+1} = \frac{1}{2}$.

(2) The algorithm can mutate the program into a more general one by applying the *RemoveNode* rule to remove the variable d:

$$\begin{tikzcd} \overset{\textsf{a}}{\boxed{\langle[1.2, 1.2]\rangle}} & \overset{\textsf{b}}{\boxed{\langle[0.2, 0.2]\rangle}} \arrow[l] \arrow[r] & \overset{\textsf{c}}{\boxed{\langle[0.4, 0.4]\rangle}} \end{tikzcd}.$$

While its quality score ψ is still $\frac{1}{2}$, the mutated program is strictly more general ($\sqsubset$) than the previous one. Therefore, it is selected as the next program to be mutated.

(3) After five more mutations using the *RemoveNode* and *GeneralizeNode* rules, it generates:

$$\begin{tikzcd} \overset{\textsf{a}}{\boxed{\langle[-\infty, \infty]\rangle}} & \overset{\textsf{b}}{\boxed{\langle[-\infty, \infty]\rangle}} \arrow[l] \end{tikzcd}.$$

where its quality score is $\psi = \frac{|\{v_1,v_4\}|}{|\{v_1,v_4\}|+1} = \frac{2}{3}$.

(4) Now, since no more better program exists as the result of the mutation, the algorithm terminates and returns the above program as the synthesized candidate program.

In our evaluation, the top-down algorithm shows strength in discovering important feature values, while the bottom-up algorithm excels at capturing key graph structures. The detailed observations are discussed in Section E of our supplementary material.

Discussion. Our synthesis algorithms may fail to find optimal GDL programs maximizing the score ψ. This is because the two synthesis algorithms perform a greedy search; they may end up with a local optimum. We would like to note that designing a sound (i.e., synthesize optimal programs) or complete (i.e., synthesized programs are optimal) algorithm for the GDL synthesis problem is challenging because it is difficult to check whether a given GDL program is the globaly best-scored one or not in the search space.

6 EVALUATION

We implement PL4XGL in Python and evaluate our model compared to the state-of-the-art GNN explanation technique, SubgraphX [Yuan et al. 2021], to answer the following research questions:

- **RQ1 (Explanation Cost):** How much faster is PL4XGL compared to SubgraphX in producing explanations for classification results?
- **RQ2 (Correctness of Explanations):** Does PL4XGL guarantee to produce correct explanations for classification results while SubgraphX does not?
- **RQ3 (Classification Accuracy):** How accurately does PL4XGL classify graph components compared to GNN-based models while providing explanations?

We evaluate PL4XGL on graph and node classification tasks, which are two main applications of graph neural networks [Wu et al. 2021].

Datasets for Graph Classification. For graph classification, we use four real-world *molecular* datasets [Debnath et al. 1991; Wu et al. 2018]: MUTAG, BBBP, BACE, and HIV. These molecular datasets have been widely used for evaluating graph-classification models [Li et al. 2022; Xu et al. 2019; Ying et al. 2019] as well as for evaluating GNN explanation techniques [Luo et al. 2020; Ying et al. 2019; Yuan et al. 2022, 2021]. A node and an edge in the molecular datasets represent an atom and a bond of a molecule, respectively. In the MUTAG dataset, the node and edge features present the types of atoms (i.e., carbon, nitrogen, oxygen, fluorine, iodine, chlorine, and bromine) and bonds (i.e., aromatic, single, double, and triple), and the graph labels indicate mutagenicity (mutagenic effects) on Salmonella typhimurium. In the BBBP, BACE, and HIV datasets, the node features describe chemical and topological properties such as atomic numbers and degrees. Edge features present chemical properties such as bond types. The labels in the BBBP dataset indicate the blood-brain barrier permeability, where predicting the penetration is a long standing problem in development of drugs targeting central nervous system [Morofuji and Nakagawa 2020]. The labels in BACE present qualitative (binary label) binding results for a set of inhibitors of human beta-secretase 1 (BACE-1). The labels in the HIV dataset indicate the ability to inhibit HIV replication.

Datasets for Node Classification. For node classification, we use eight datasets in three different domains: 1) two for *synthetic* datasets [Yuan et al. 2022], 2) three for *web page* datasets [Pei et al. 2020], and three for *citation network* datasets [Wu et al. 2021]. The two *synthetic* datasets BA-Shapes and Tree-Cycles also have been widely used for evaluating and comparing various GNN explanation methods [Luo et al. 2020; Ying et al. 2019; Yuan et al. 2022, 2021]. (1) The BA-Shapes dataset is constructed with a base Barabási-Albert (BA) graph on 300 nodes, and 80 house-structured motifs. Each house-structured motif consists of five nodes and is attached to a node randomly selected from

Table 1. Statistics of the datasets

	Graph classification				Node classification							
	Molecular datasets				Synthetic datasets		Web page datasets			Citation networks		
	MUTAG	BBBP	BACE	HIV	BA-Shapes	Tree-Cycles	Wisconsin	Texas	Cornell	Cora	Citeseer	PubMed
# Graphs	188	2,039	1,513	41,127	1	1	1	1	1	1	1	1
# Nodes (avg)	17.9	24.0	34.0	25.5	700	871	183	183	251	2,708	3,327	19,717
# Edges (avg)	19.7	25.9	36.8	27.5	2,055	971	450	279	277	5,278	4,552	44,324
# Labels	2	2	2	2	4	2	5	5	5	7	6	3
# Node features	1	9	9	9	1	1	1,703	1,703	1,703	1,433	3,703	500
# Edge features	1	3	3	3	0	0	0	0	0	0	0	0

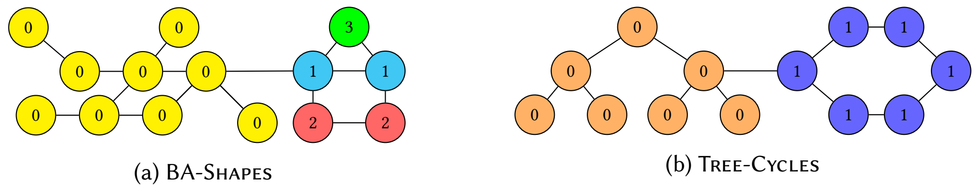


Fig. 8. Simplified examples of synthetic datasets. Numbers in nodes represent the labels nodes belong to.

the base BA graph. Additionally, 70 random edges are added to the resulting graph for perturbation. There are four different node labels based on their structural roles. Nodes that do not belong to house-structured motifs are assigned label 0, and each house motif consists of a top node (label 3), two middle nodes (label 1), and two bottom nodes (label 2). A middle node in a motif is connected to the base BA graph. Figure 8a shows an example of a house-structured motif attached to a base graph and describes how nodes are labeled differently. (2) The Tree-Cycles dataset consists of a base 8-level balanced binary tree (label 0) and 80 six-node cycle motifs (label 1). Each cycle motif is randomly attached to the base binary tree, and 87 edges are randomly added to the resulting graph. In both BA-Shapes and Tree-Cycles, we used degrees (i.e., # of edges of a node) as a node feature. In the *web page* datasets (Wisconsin, Texas, and Cornell), nodes and edges represent web pages and hyperlinks, respectively, and the labels represent the categories (i.e., student, project, course, staff, and faculty). In the *citation networks* (Cora, Citeseer, and PubMed), nodes and edges represent documents and citation links, respectively, and the labels represent document classes. In these six datasets, nodes are associated with bag-of-words feature vectors.

Baseline GNN-based Models for SubgraphX. For a fair comparison with SubgraphX, we use the original experimental setting of SubgraphX as much as possible. The baseline GNN-based models for SubgraphX are GIN [Xu et al. 2019] and GCN [Kipf and Welling 2017] because they are used for evaluation in the original paper of SubgraphX and also the most popular models on which other recent GNN explanation techniques have been evaluated [Luo et al. 2020; Ying et al. 2019; Yuan et al. 2021]. For graph classification tasks, we use GCN for BBBP and GIN for MUTAG, BACE, and HIV as the baseline GNN-based model. For node classification tasks, we use GCN as the baseline GNN-based model for all five datasets. We split the datasets into 8:1:1 for training, validation, and testing sets, respectively. For the two *molecular* datasets MUTAG and BBBP, we utilize the data split used in Yuan et al. [2021]. We applied a random split to the other remaining datasets. In our evaluation, PL4XGL used an AMD Ryzen Threadripper 3990X with 64 cores. The GNNs and SubgraphX used an NVIDIA RTX A6000 GPU for training and producing the explanations.

6.1 RQ1. Explanation Cost

We first compare the explanation cost of PL4XGL and SubgraphX [Yuan et al. 2021]. We used the artifact of Liu et al. [2021] that provides the implementation of SubgraphX. For a thorough comparison, we also compare the training and classification costs.

Table 2. Cost comparison between PL4XGL and the baseline GNN with SubgraphX in minutes. The rows "Training" show the cost of training the model, "Classification" show the cost of classifying the test sets, and "Explanation" show the cost of producing explanations for the test sets. The rows "Total" show the sum of the three costs. The bold numbers indicate the better (i.e., lower) ones. 'timeout' indicates that the method failed to finish its task within the time budget (2 days for training, 1 day for classification, and 4 days for explanation).

Dataset	Cost (minutes)	GNN+ SubgraphX	PL4XGL	Dataset	Cost (minutes)	GNN+ SubgraphX	PL4XGL
MUTAG	Training	**0.2**	12.3	Wisconsin	Training	**0.4**	8.0
	Classification	**0.1**	**0.1**		Classification	**0.1**	**0.1**
	Explanation	8.4	**0.0**		Explanation	69.3	**0.0**
	Total	**8.7**	12.4		Total	69.5	**8.1**
BBBP	Training	**1.0**	34.3	Texas	Training	**0.4**	5.0
	Classification	**0.1**	0.7		Classification	**0.1**	**0.1**
	Explanation	160.0	**0.0**		Explanation	52.1	**0.0**
	Total	161.1	**35.0**		Total	52.3	**5.1**
BACE	Training	**1.0**	60.6	Cornell	Training	**0.3**	5.0
	Classification	**0.1**	4.0		Classification	**0.1**	**0.1**
	Explanation	141.1	**0.0**		Explanation	95.8	**0.0**
	Total	142.2	**69.9**		Total	96.0	**5.1**
HIV	Training	12.2	timeout	Cora	Training	**0.4**	61.6
	Classification	0.1	N/A		Classification	**0.1**	0.9
	Explanation	2887.8	N/A		Explanation	timeout	**0.0**
	Total	2900.1	timeout		Total	timeout	62.5
BA-Shapes	Training	**0.1**	0.2	Citeseer	Training	**0.4**	245.2
	Classification	**0.1**	**0.1**		Classification	**0.1**	2.0
	Explanation	4756.0	**0.0**		Explanation	timeout	**0.0**
	Total	4756.2	**0.2**		Total	timeout	**247.2**
Tree-Cycles	Training	**0.1**	0.2	Pubmed	Training	**0.6**	2702.9
	Classification	**0.1**	**0.1**		Classification	**0.1**	17.0
	Explanation	3.4	**0.0**		Explanation	timeout	**0.0**
	Total	3.6	**0.2**		Total	timeout	**2719.9**

Table 2 shows the cost comparison in minutes. The columns "GNN+SubgraphX" and "PL4XGL" represent the cost of the baseline and PL4XGL, respectively. The rows "Training", "Classification", and "Explanation" present the respective costs for training the model, classifying the test sets, and generating explanations for the test sets. The rows "Total" sum the three costs. The bold numbers indicate the better (i.e., lower) ones. The term "timeout" means the method failed to finish its task within the time budget. The time budget was 2 days for training, 1 day for classification, and 4 days for generating explanations.

As PL4XGL is designed to produce explanations for predictions simultaneously, its explanation cost is always 0. By contrast, the explanation cost of SubgraphX was significant. For the BA-Shapes dataset, SubgraphX took about 3 days to produce explanations for the test set. SubgraphX took this amount of time because it explored a huge number of candidate subgraphs. For example, to explain a single node belonging to the Barabási-Albert graph (label 0), SubgraphX had to explore 2^{355} candidate subgraphs. Note that SubgraphX failed to produce explanations for the three *citation network* datasets because of its expensive explanation cost. In terms of classification and explanation costs (i.e.,"Classification" + "Explanation"), PL4XGL was at least 35 times faster.

The classification and training costs, however, show a trade-off of PL4XGL. In the largest dataset HIV, for example, PL4XGL failed to finish its learning within the time budget (i.e., 2 days), whereas the baseline finished its learning within 12.2 minutes. In the PUBMED dataset, the classification task took 17.0 minutes in PL4XGL, while the classification cost of the baseline was negligible. However, we would like to note that except for the MUTAG and HIV datasets, PL4XGL outperforms the baseline in terms of the "Total" cost.

6.2 RQ2. Correctness of Explanations

Now, we compare the correctness (i.e., whether the provided explanations reflect the actual reasons for the classifications) of those explanations produced by PL4XGL and SUBGRAPHX with widely used metrics. For evaluating the quality of GNN explanation techniques, various metrics have been proposed [Kakkad et al. 2023; Yuan et al. 2022]. Unfortunately, however, there is no metric that can directly measure the correctness of the explanations because the actual reasons for the classifications are unavailable in black-box models (GNNs). Instead, *Fidelity*, which can be seen as a proxy metric for measureing the correctness, has been widely used [Lucic et al. 2022].

Fidelity is designed for quantifying the faithfulness of subgraph explanations to the underlying model, and variations of *Fidelity* exist [Yuan et al. 2022]. In our evaluation, we employ *Fidelity–*^{acc}, which is applicable to both PL4XGL and SUBGRAPHX. The insight behind the metric *Fidelity–*^{acc} is that if a provided explanation subgraph is the actual reason for a prediction on an original graph, then the model ought to classify the subgraph into the same label. *Fidelity–*^{acc} assesses whether the model's predictions for the subgraphs are identical to the original ones as follows:

$$\textit{Fidelity–}^{acc} = \frac{1}{N}\sum_{i=1}^{N}(\mathbb{1}(\hat{y}_i = y_i) - \mathbb{1}(\hat{y}_i^{m_i} = y_i)) \qquad \text{(lower is better).}$$

N is the number of explained classifications, y_i represents an original classification result (for the i^{th} component in the test set), $\hat{y}_i$ is the classification result for the original graph. m_i presents the nodes in the explanation subgraph, and $\hat{y}_i^{m_i}$ presents the classification result for the subgraph. The indicator function $\mathbb{1}(a = b)$ equals 1 if a and b are the same, and 0 otherwise. Then, the equation $\mathbb{1}(\hat{y}_i = y_i) - \mathbb{1}(\hat{y}_i^{m_i} = y_i)$ equals 0 if the model produces the same label for the given subgraph, and 1 otherwise. In *Fidelity–*^{acc}, a lower score indicates the greater faithfulness. For simplicity, we denote *Fidelity–*^{acc} as *Fidelity*.

Sparsity is typically evaluated in conjunction with *Fidelity*. Evaluating *Fidelity* alone is insufficient for comparing explanations, as using the original graphs as subgraph explanations would result in an optimal *Fidelity* score of 0. The key insight behind *Sparsity* is that effective explanations should be sparse and simple; smaller subgraphs are considered better. The formal definition of *Sparsity* is:

$$\textit{Sparsity} = \frac{1}{N}\sum_{i=1}^{N}(1 - \frac{|m_i|}{|M_i|}) \qquad \text{(higher is better)}$$

where $|m_i|$ and $|M_i|$[3] denote the number of nodes in the explanation subgraph and the original graph, respectively. In *Sparsity*, the higher is the better, implying greater simplicity.

As the two metrics above are designed to compare subgraph explanations, we translated the GDL program explanations of PL4XGL into subgraph explanations. Figure 9 shows how a GDL program is translated into a subgraph. The column "Original graph" presents a graph G in the MUTAG dataset classified by PL4XGL. The column "GDL program" denotes the provided explanation program $P = \overline{\delta}$ `target graph` for the classification. The column "Transformed subgraph" presents a transformed subgraph $G' \in \{G|_\eta \mid (G, \eta) \in [\![\overline{\delta}]\!]\}$. In the explanation, the black colored edges

[3]In node classification datasets where the size of the original graph is large (e.g., #nodes is 700 in a graph), we define M_i as the number of nodes within two (WISCONSIN, TEXAS, and CORNELL) or three hops (BA-SHAPES and TREE-CYCLES) from the target explained node. This adjustment is for more meaningful comparison of explanation sparsity [Yuan et al. 2021].

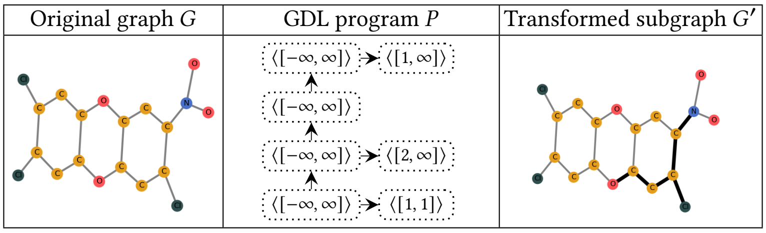

Original graph G
GDL program P
Transformed subgraph G'
$\langle[-\infty,\infty]\rangle$
$\langle[1,\infty]\rangle$
$\langle[2,\infty]\rangle$
$\langle[1,1]\rangle$

Fig. 9. How we translate a GDL program into a subgraph explanation.

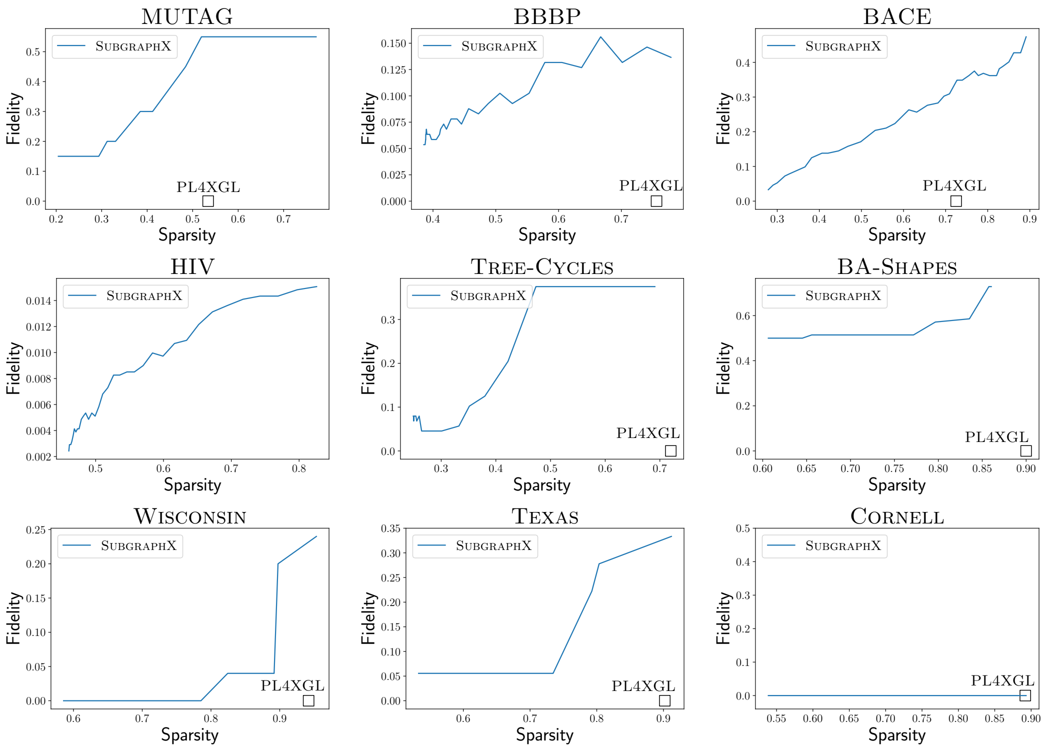


Fig. 10. Comparison of *Fidelity* and *Sparsity* between PL4XGL and SubgraphX. The size of the subgraph explanations is parameterized in SubgraphX; the blue lines show how *Fidelity* and *Sparsity* changes over the chosen hyperparameter values (i.e., size of the explanations). In PL4XGL, the size of explanation is determined by the model. The squares (□) present *Fidelity* and *Sparsity* of PL4XGL's explanations.

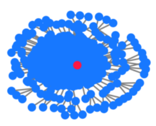
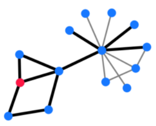
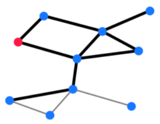
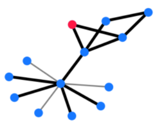
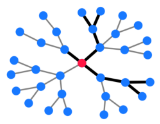
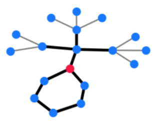
represent edge descriptions with the most general constraint $\langle[-\infty,\infty]\rangle$; they describe all the types of bonds. The column "Transformed subgraph" presents a subgraph (bold edges) described by the GDL program P. For example, the node descriptions $\langle[-\infty,\infty]\rangle$, $\langle[1,1]\rangle$, $\langle[2,\infty]\rangle$, $\langle[1,\infty]\rangle$ are valuated as carbon (C), nitrogen (N), chlorine (cl), and oxygen (O). For a given graph G, multiple subgraphs of it can be described by a GDL program P (i.e., $|\{G|_\eta \mid (G,\eta) \in [\![\bar{\delta}]\!]\}| > 1$). However, all the subgraphs in $\{G|_\eta \mid (G,\eta) \in [\![\bar{\delta}]\!]\}$ have the same *Sparsity* score because the subgraphs have the same number of nodes. Also, they are guaranteed to achieve the same *Fidelity* score (Theorem 6.1).

Figure 10 compares *Fidelity* and *Sparsity* of PL4XGL and SubgraphX for the nine datasets. *Fidelity* and *Sparsity* of SubgraphX are not available for the *citation networks* as it failed to produce explanations. Similarly, these metrics of PL4XGL are not available for the HIV dataset. In the plots,

Table 3. *Fidelity* and *Sparsity* score of PL4XGL for the *citation networks*. As SubgraphX failed to produce explanations within the time budget (i.e., four days), the values are unavailable (N/A).

	Fidelity			*Sparsity*		
	Cora	Citeseer	Pubmed	Cora	Citeseer	Pubmed
PL4XGL	0.0	0.0	0.0	0.82	0.66	0.92
SubgraphX	N/A	N/A	N/A	N/A	N/A	N/A

Table 4. Explanations produced by PL4XGL and SubgraphX for the synthetic datasets.

	BA-Shapes				Tree-Cycles	
	Label 0	Label 1	Label 2	Label 3	Label 0	Label 1
PL4XGL	$\langle[12,\infty]\rangle \rightarrow \langle[-\infty,\infty]\rangle \leftarrow \langle[12,\infty]\rangle$	$\langle[4,\infty]\rangle \rightarrow \langle[3,4]\rangle \leftarrow \langle[2,2]\rangle$	$\langle[2,2]\rangle \rightarrow \langle[2,2]\rangle$	$\langle[4,\infty]\rangle \rightarrow \langle[2,2]\rangle \leftarrow \langle[3,4]\rangle$	$\langle[3,5]\rangle \leftarrow \langle[3,3]\rangle \leftarrow \langle[3,3]\rangle$	$\langle[-\infty,\infty]\rangle \leftarrow \langle[-\infty,2]\rangle \leftarrow \langle[-\infty,2]\rangle$
SubgraphX						

the Y-axis represents *Fidelity*, with lower values indicating better, more faithful explanations. The X-axis denotes *Sparsity*, where higher values correspond to simpler (and thus better) explanations. In the plots, the squares and blue lines represent the performance of PL4XGL and SubgraphX, respectively. When explaining a prediction, SubgraphX provides a subgraph that can include at most k nodes where the value of k is a hyperparameter given by users. In our experiments, we used k values ranging from 5 to 40 for graph classification and from 1 to 30 for node classification. Unlike SubgraphX, the *Fidelity* and *Sparsity* values of PL4XGL are fixed as the model itself chooses the size of the explanations.

Figure 10 demonstrates that PL4XGL provides significantly more faithful yet sparse explanations compared to SubgraphX. PL4XGL outperforms SubgraphX in terms of *Fidelity* for all datasets, achieving the optimal score of 0. We would like to note that *Fidelity* of PL4XGL is guaranteed to be 0 for any dataset. In graph classifications, for example, the following theorem holds.[4]

Theorem 6.1. *If PL4XGL classifies a graph G into a label i and provides a GDL program P as an explanation, PL4XGL classifies all the subgraphs transformed from P into the same label i.*

The explanations of PL4XGL are also sparse. The transformed subgraphs consist of about only 21% of the nodes in the original graphs. PL4XGL also provided simple (high *Sparsity* score) and model-faithful (low *Fidelity* score) explanations for the *citation networks* as described in Table 3. The explanations of PL4XGL are also general that can be applicable to a more number of predictions compared to SubgraphX. Due to the space limit, the detailed results are presented and discussed in Section D of our supplementary material.

Qualitative Comparison on the Synthetic Datasets. In the synthetic datsets, where the labels have certain properties, the quality of the explanations can therefore be measured by checking whether the properties are identified or not. The rows "PL4XGL" and "SubgraphX" in Table 4 present the explanations provided by PL4XGL and SubgraphX. Each column presents the explanation for a label in the datasets.

[4]See Section B of our supplementary material for the proof and the corresponding theorem for node classification.

Table 5. Classification accuracy (%) comparison of PL4XGL against representative GNNs.

	GCN	GAT	ChebyNet	JKNet	GraphSAGE	GIN	DGCN	PL4XGL
MUTAG	80.0±0.0	89.0±2.2	86.0±4.1	68.0±7.5	78.0±4.4	91.0±5.4	N/A	**100.0±0.0**
BBBP	83.6±1.4	82.3±1.6	84.6±1.0	85.6±1.9	86.6±0.9	86.2±1.4	N/A	**86.8±0.0**
BACE	78.4±2.8	52.4±3.3	78.9±1.4	79.9±1.9	79.8±0.8	**80.9±0.4**	N/A	**80.9±0.0**
HIV	96.4±0.0	96.4±0.0	96.8±0.2	96.8±0.1	**96.9±0.2**	96.8±0.1	N/A	N/A
BA-Shapes	95.1±0.6	76.8±2.3	**97.1±0.0**	94.3±0.0	**97.1±0.0**	92.0±1.1	95.1±0.7	95.7±0.0
Tree-Cycles	97.7±0.0	90.9±0.0	**100.0±0.0**	98.9±0.0	**100.0±0.0**	93.2±0.0	99.2±0.5	**100.0±0.0**
Wisconsin	64.0±0.0	49.6±3.1	86.4±3.9	64.8±1.5	92.8±2.9	56.0±0.0	**96.0±0.0**	88.0±0.0
Texas	67.7±5.3	50.0±0.0	87.7±2.1	68.8±4.3	86.6±2.6	50.0±0.0	**86.6±2.6**	83.3±0.0
Cornell	58.9±2.6	61.1±0.0	81.0±6.5	61.1±0.0	87.7±2.1	61.1±0.0	86.6±2.6	**88.8±0.0**
Cora	85.6±0.3	86.4±1.8	86.5±5.2	84.9±3.5	86.3±3.2	**86.7±0.0**	83.2±5.9	80.0± 0.0
Citeseer	75.2±0.0	74.3±0.7	**79.1±0.9**	73.7±4.2	75.9±2.3	75.2±0.0	71.3±6.0	63.8± 0.0
Pubmed	82.8±1.1	84.7±1.2	**88.7±1.0**	83.2±0.4	88.0±0.4	86.1±0.6	85.1±0.6	81.4±0.0

The GDL programs in Table 4 describe the properties of the synthetic datasets. For example, $\langle[12,\infty]\rangle \rightarrow \langle[-\infty,\infty]\rangle \leftarrow \langle[12,\infty]\rangle$ presents a property of a Barabási-Albert (BA) graph (i.e., label 0 in BA-Shapes), which is often used to model several human-made networks (e.g., world wide web), a majority of nodes (e.g., web pages) are connected with other nodes that have a large number of edges. The program precisely and robustly desribed nodes in the Barabási-Albert graphs. In the BA-Shapes dataset, 97% of the nodes in label 0 are described by the program, and 99% of the nodes belong to the program have the label 0. $\langle[2,2]\rangle \leftarrow \langle[2,2]\rangle$ describes a property of a bottom node (label 2) in house motifs that has two edges and is connected to another bottom node. $\langle[4,\infty]\rangle \rightarrow \langle[3,4]\rangle \leftarrow \langle[2,2]\rangle$ explains that the middle nodes (label 1) have three or four edges and can be connected to any type of nodes (e.g., top, middle, bottom, and BA nodes), where $\langle[2,2]\rangle$ describes a top or bottom node, and $\langle[4,\infty]\rangle$ a middle node or a node in the BA graph (that has at least four edges). $\langle[4,\infty]\rangle \rightarrow \langle[2,2]\rangle \leftarrow \langle[3,4]\rangle$ captures that top nodes (label 3) have two edges and are connected with two middle nodes who have three or four edges. $\langle[3,4]\rangle$ and $\langle[4,\infty]\rangle$ capture the middle nodes that have three and four nodes, respectively. In the Tree-Cycles dataset, $\langle[3,5]\rangle \leftarrow \langle[3,3]\rangle \leftarrow \langle[3,3]\rangle$ describes a property of internal nodes in a binary tree (label 0). $\langle[-\infty,\infty]\rangle \leftarrow \langle[-\infty,2]\rangle \leftarrow \langle[-\infty,2]\rangle$ describes a property of the nodes in Cycle motifs (label 1) that have an adjacent node that has two edges, and the adjacent node also has another adjacent node that has two edges.

The subgraph explanations of SubgraphX capture the key subgraphs (i.e., motifs) for the labels. The subgraphs (bold edges) in Table 4 explain why the red-colored nodes are classified into the corresponding labels. For example in Label 1 of the Tree-Cycles dataset, the explanation illustrates that the node is classified into Label 1 because the node is in a cycle motif.

6.3 RQ3. Classification Accuracy

Now, we compare the classification accuracy of PL4XGL against representative GNNs.

Baseline GNNs. We evaluate the classification accuracy of PL4XGL in comparison with seven graph neural networks: GCN [Kipf and Welling 2017], GAT [Veličković et al. 2018], ChebyNet [Defferrard et al. 2016], JKNet [Xu et al. 2018], GraphSage [Hamilton et al. 2017], GIN [Xu et al. 2019], and DGCN [Park et al. 2022]. GCN, GAT, GIN are provided by the artifact of SubgraphX [Yuan et al. 2021], and we additionally implemented ChebyNet, JKNet, and GraphSage. We chose these GNNs (except for the last) from a recent survey [Wu et al. 2021], which introduces representative GNN models. For comparison with more recent GNNs, we include DGCN [Park et al. 2022], which addressed limitations of GNNs in classifying heterophilic graphs such as the graphs in the *web page* datasets. Details of training GNNs are described in Section C of our supplementary material.

Table 6. How the learning cost and accuracy change with respect to the value of k in *Mutate*^{k}.

		Mutate	*Mutate*^{2}	*Mutate*^{3}	*Mutate*^{4}	*Mutate*^{5}	*Mutate*^{6}	*Mutate*^{7}	*Mutate*^{8}
BA-Shapes	Training cost (s)	1	2	8	19	39	100	392	1845
	Accuracy	90	95.7	95.7	94.2	95.7	95.7	95.7	95.7
Tree-Cycles	Training cost (s)	1	3	5	10	20	63	108	1492
	Accuracy	70.4	100	100	100	98.8	98.8	98.8	98.8

Classification Accuracy. Table 5 compares the classification accuracy, which reports the mean accuracy over five runs along with the corresponding 95% confidence intervals. Since learning GNNs is affected by random seeds, the five runs may show different performance. PL4XGL, however, does not use such random seeds; the five runs produced the same accuracy. The results demonstrate that PL4XGL achieves competitive accuracy in comparison to the baseline GNN models. The classification accuracy of DGCN is not available for the graph classification datasets as DGCN is designed for node classification tasks [Park et al. 2022]. For the five datasets MUTAG, BBBP, BACE, Tree-Cycles, and Cornell, PL4XGL shows the best accuracy. In the three datasets BA-Shapes, Wisconsin, and Texas, PL4XGL shows the third or fourth-best accuracy. In the three citation network datasets, PL4XGL achieved the worst accuracy, and it shows a limitation of our approach discussed in Section 7. We would like to note that PL4XGL shows the best accuracy for the three *molecular* datasets, which are decision-critical datasets related to the development of safe drugs.

Qualitative Analysis. PL4XGL achieved the high accuracy for the five datasets MUTAG, BBBP, BACE, Tree-Cycles, and Cornell thanks to the learned high-quality GDL programs. For instance, the following GDL program contributed significantly to the high accuracy in the MUTAG dataset:

$$\begin{tikzcd}
 & & \langle[-\infty,\infty]\rangle & & & & & \\
\langle[-\infty,\infty]\rangle \arrow[r] & \langle[0,0]\rangle \arrow[r] & \langle[-\infty,\infty]\rangle \arrow[u] \arrow[r] & \langle[-\infty,\infty]\rangle \arrow[r] & \langle[-\infty,\infty]\rangle \arrow[r] & \langle[-\infty,\infty]\rangle \arrow[r] & \langle[-\infty,\infty]\rangle \arrow[r] & \langle[-\infty,\infty]\rangle
\end{tikzcd}$$

In the GDL program, the black colored edges and nodes with the constraint $\langle[-\infty,\infty]\rangle$ denote any type of bonds (e.g., aromatic, single, double, triple) and atoms (e.g., carbon, nitrogen, oxygen, fluorine, iodine, chlorine, bromine), respectively. The green colored edges and the nodes with the constraint $\langle[0,0]\rangle$ represent aromatic bonds and carbons, respectively. In the classification task, the above GDL program classified 45% of the test graphs, all of which were correctly classified. We guess the above GDL program captures a key pattern of the mutagenic effects on Salmonella typhimurium. In the MUTAG dataset, the above GDL program describes only the mutagenetic molecules (i.e., precision = 100%), and 77% of the mutegenic molecules are described by the above GDL program (i.e., recall = 77%).

Impact of the enumeration depth. Table 6 shows how the value of k in *Mutate*^{k} (from Algorithm 1) affected the training cost and accuracy on the two synthetic datasets. In Table 6, the rows "Training cost (s)" present the training cost in seconds, and the rows "Accuracy" present the classification accuracy for the test sets. The results show that using the smallest k (i.e., $k = 1$) was insufficient for achieving high accuracy in the two synthetic datasets. This was because the search algorithm failed to discover high-quality GDL programs with the smallest search depth. We would like to note that we used $k = 1$ for the ten real-world datasets because using a larger k (e.g., $k = 2$) failed its learning within the time budget. However, the results also indicate that using an excessively large k (e.g., $k = 8$) may lower the accuracy. In the Tree-Cycles dataset, the accuracy decreased from 100% to 98.8% when k increased from 2 to 8. This was because the algorithm synthesized an overfitted GDL programs when it used the large search depth.

7 DISCUSSION

This section discusses limitations of PL4XGL and future work. In addition, we compare the expressiveness of GDL against subgraphs as a graph pattern description language.

7.1 Limitations and Future Work

PL4XGL has inherent strength in explainability, but it currently has many limitations and much room for improvement.

Limited Expressiveness of GDL. The current language is not expressive enough to describe diverse graph properties; GDL may not capture key properties of datasets, which may lead to suboptimal accuracy. For example, PL4XGL showed relatively lower accuracy for the *citation network* datasets (Cora, Citeseer, Pubmed) because the current GDL is unable to describe key properties of the datasets. *Citation networks* are homophilic graphs (i.e., nodes in the same labels are usually connected), and GNNs are particularly effective for such datasets because they are designed under the assumption that input graphs are homophilic [Park et al. 2022]. However, PL4XGL failed to achieve high accuracy for the datasets because the current GDL is unable to describe this homophilic property. There exist many other properties that the current GDL is unable to describe (e.g., "molecules containing more oxygens than chlorines"). Therefore, enhancing the expressiveness of GDL is a promising research direction, potentially improving the performance of PL4XGL across diverse datasets.

Expensive Training and Classification Costs. Though the explanation cost is removed, PL4XGL requires significantly higher training and classification costs compared to the baseline GNNs. As described in Table 2, PL4XGL failed its learning in the HIV dataset, and the classification cost of PL4XGL is 170 times higher than the baseline GNN in the Pubmed dataset. The expensive training and classification costs mainly come from the current approach generating too many GDL programs. We would like to note that the model used only a few learned GDL programs for the classification task. In the MUTAG dataset, for example, PL4XGL used only 5% of the learned GDL programs. Therefore, if the training process could be optimized to learn and retain only the essential GDL programs, training and classification costs could be significantly reduced.

7.2 Expressiveness Comparison between Subgraph and GDL

As a graph pattern description language, GDL is strictly more expressive than subgraphs. That is, a subgraph can be represented by a GDL program, but not vice versa. For example, the following subgraph G' describes a graph pattern that three nodes are connected by two edges where each node and edge has the same feature vector $\langle 1.0\rangle$:

$$\begin{tikzcd} G' = \overset{v_1}{\boxed{\langle 1.0\rangle}} \arrow[r, "\langle 1.0\rangle"] & \overset{v_2}{\boxed{\langle 1.0\rangle}} \arrow[r, "\langle 1.0\rangle"] & \overset{v_3}{\boxed{\langle 1.0\rangle}} . \end{tikzcd}$$

The above subgraph can be described by the following GDL program P:

$$\begin{tikzcd} P = \overset{\text{x}}{\boxed{\langle [1.0, 1.0]\rangle}} \arrow[r, "\langle [1.0, 1.0]\rangle"] & \overset{\text{y}}{\boxed{\langle [1.0, 1.0]\rangle}} \arrow[r, "\langle [1.0, 1.0]\rangle"] & \overset{\text{z}}{\boxed{\langle [1.0, 1.0]\rangle}} . \end{tikzcd}$$

The subgraph G' and the GDL program P above describe the same graph pattern. However, the following GDL program P', where each node and edge variable has a feature value constraint $\langle [0.0, 5.0]\rangle$, cannot be equivalently described by a single subgraph:

$$\begin{tikzcd} P' = \overset{\text{x}}{\boxed{\langle [0.0, 5.0]\rangle}} \arrow[r, "\langle [0.0, 5.0]\rangle"] & \overset{\text{y}}{\boxed{\langle [0.0, 5.0]\rangle}} \arrow[r, "\langle [0.0, 5.0]\rangle"] & \overset{\text{z}}{\boxed{\langle [0.0, 5.0]\rangle}} . \end{tikzcd}$$

This is because nodes and edges in a subgraph are associated with concrete feature values, unlike the more flexible representations possible with GDL.

8 RELATED WORK

In this section, we discuss previous work closely related to ours.

Improving Explainability of Graph Neural Networks. Instead of developing a new method for explainable graph learning, numerous works have focused on improving explainability of graph neural networks [Feng et al. 2022a; Funke et al. 2021; Lucic et al. 2022; Pope et al. 2019; Schnake et al. 2021; Vu and Thai 2020a; Wu et al. 2022; Ying et al. 2019; Zhang et al. 2021, 2022]. GraphMask [Schlichtkrull et al. 2021] and PGExplainer [Luo et al. 2020] learn a classifier that predicts whether the removal of an edge would affect the classification results; they use the learned classifier to identify important subgraphs. GraphLime [Huang et al. 2022] identifies important node features by fitting a feature selection algorithm, Hilbert-Schmidt Independence Criterion Lasso, to local classification results of the given GNNs. Similarly, PGM-Explainer [Vu and Thai 2020b] uses bayesian network to explain local classification results of GNNs. KerGNN [Feng et al. 2022b] classifies a node with its pre-trained graphs, named graph filters, using graph kernels that measure the similarity of the subgraph of the node and the graph filters. The used graph filters can be provided as explanations. Instead of providing instance-level explanations, XGNN provides model-level explanations. XGNN [Yuan et al. 2020] uses reinforcement learning to generate graph patterns that maximize certain prediction of the given GNN.

Graph Pattern Description Languages. In the literature, subgraphs have been used as a dominant graph pattern description language; GDL can be employed instead of subgraphs. For example, graph data mining and GNN explanation techniques [Kakkad et al. 2023; Ramraj and Prabhakar 2015] have produced valuable subgraphs in graph datasets. Inokuchi et al. [2000] use the Apriori-based algorithm to mine frequent subgraphs. The Apriori-based algorithm searches frequent patterns from a simple one to a complex one like our top-down synthesis algorithm. Yan and Han [2002] use a pattern growth approach that employs DFS to enumerate possible subgraphs. Existing GNN explanation techniques also try to find valuable subgraphs [Ye et al. 2023] for explaining predictions of GNNs. As discussed in Section 7.2, GDL is strictly more expressive than subgraphs; GDL can be employed in graph data mining and GNN explanation techniques. For example, $\langle[12,\infty]\rangle \rightarrow \boxed{\langle[-\infty,\infty]\rangle} \leftarrow \langle[12,\infty]\rangle$ precisely (precision = 99%) and robustly (recall = 97%) describes a key property of nodes in the Barabási-Albert graph thanks to the abstract feature values $[12,\infty]$ and $[-\infty,\infty]$, which are unavailable in the subgraph-based graph pattern descriptions.

9 CONCLUSION

In this paper, we investigated a new approach to accurate and explainable machine learning on graphs. Deviating from the mainstream approaches based on GNNs, we developed a new graph learning approach using two programming language techniques. First, we designed a domain-specific language, GDL, for interpretable graph learning models; GDL programs serve as the source code for our models. Second, we formulated the learning problem as a GDL program synthesis problem and adapted two representative synthesis algorithms to learn GDL programs from training data. The experimental results showed that our approach accurately classifies graph data and provides correct explanations. We believe that our work can pave the way for future work that will be both fascinating and useful in applying programming language techniques to graph learning (and beyond).

DATA-AVAILABILITY STATEMENT

The artifact of PL4XGL is available in Zenodo [Jeon 2024] and GitHub[5]. The artifact includes the implementation of PL4XGL, the datasets used in the evaluation, and the evaluation scripts.

ACKNOWLEDGMENTS

We thank Seunghun Lee and Seongjun Yun for their helpful comments on the baseline GNNs. This work was partly supported by Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government(MSIT) (No.2020-0-01337,(SW STAR LAB) Research on Highly-Practical Automated Software Repair) and by ICT Creative Consilience Program through the Institute of Information & Communications Technology Planning & Evaluation(IITP) grant funded by the Korea government(MSIT) (IITP-2024-2020-0-01819, 10%). This work was also supported by the National Research Foundation of Korea(NRF) grant funded by the Korea government(MSIT)(No. 2021R1A5A1021944) and Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government(MSIT) (No.2021-0-00177, High Assurance of Smart Contract for Secure Software Development Life Cycle, No.2022-0-00277, Development of SBOM Technologies for Securing Software Supply Chains). Jihyeok Park and Hakjoo Oh are corresponding authors.

REFERENCES

Rajeev Alur, Arjun Radhakrishna, and Abhishek Udupa. 2017. Scaling Enumerative Program Synthesis via Divide and Conquer. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. https://www.microsoft.com/en-us/research/publication/scaling-enumerative-program-synthesis-via-divide-and-conquer/

Rajeev Alur, Rishabh Singh, Dana Fisman, and Armando Solar-Lezama. 2018. Search-Based Program Synthesis. *Commun. ACM* 61, 12 (nov 2018), 84–93. https://doi.org/10.1145/3208071

Asim Kumar Debnath, Rosa L. Lopez de Compadre, Gargi Debnath, Alan J. Shusterman, and Corwin Hansch. 1991. Structure-activity relationship of mutagenic aromatic and heteroaromatic nitro compounds. Correlation with molecular orbital energies and hydrophobicity. *Journal of Medicinal Chemistry* 34, 2 (1991), 786–797. https://doi.org/10.1021/jm00106a046 arXiv:https://doi.org/10.1021/jm00106a046

Michaël Defferrard, Xavier Bresson, and Pierre Vandergheynst. 2016. Convolutional Neural Networks on Graphs with Fast Localized Spectral Filtering. In *Proceedings of the 30th International Conference on Neural Information Processing Systems* (Barcelona, Spain) *(NIPS'16)*. Curran Associates Inc., Red Hook, NY, USA, 3844–3852.

Elizabeth Dinella, Hanjun Dai, Ziyang Li, Mayur Naik, Le Song, and Ke Wang. 2020. HOPPITY: LEARNING GRAPH TRANSFORMATIONS TO DETECT AND FIX BUGS IN PROGRAMS. In *International Conference on Learning Representations*. https://openreview.net/forum?id=SJeqs6EFvB

Aosong Feng, Chenyu You, Shiqiang Wang, and Leandros Tassiulas. 2022b. KerGNNs: Interpretable Graph Neural Networks with Graph Kernels. *Proceedings of the AAAI Conference on Artificial Intelligence* 36, 6 (Jun. 2022), 6614–6622. https://doi.org/10.1609/aaai.v36i6.20615

Jiarui Feng, Lecheng Kong, Hao Liu, Dacheng Tao, Fuhai Li, Muhan Zhang, and Yixin Chen. 2023. Extending the Design Space of Graph Neural Networks by Rethinking Folklore Weisfeiler-Lehman. In *Advances in Neural Information Processing Systems*.

Qizhang Feng, Ninghao Liu, Fan Yang, Ruixiang Tang, Mengnan Du, and Xia Hu. 2022a. DEGREE: Decomposition Based Explanation for Graph Neural Networks. In *International Conference on Learning Representations*. https://openreview.net/forum?id=Ve0Wth3ptT_

John K. Feser, Swarat Chaudhuri, and Isil Dillig. 2015. Synthesizing Data Structure Transformations from Input-Output Examples. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Portland, OR, USA) *(PLDI '15)*. Association for Computing Machinery, New York, NY, USA, 229–239. https://doi.org/10.1145/2737924.2737977

Jonathan Frankle, Peter-Michael Osera, David Walker, and Steve Zdancewic. 2016. Example-Directed Synthesis: A Type-Theoretic Interpretation. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (St. Petersburg, FL, USA) *(POPL '16)*. Association for Computing Machinery, New York, NY, USA, 802–815. https://doi.org/10.1145/2837614.2837629

[5]https://github.com/kupl/PL4XGL

Thorben Funke, Megha Khosla, and Avishek Anand. 2021. Hard Masking for Explaining Graph Neural Networks. https://openreview.net/forum?id=uDN8pRAdsoC

Sumit Gulwani. 2011. Automating String Processing in Spreadsheets Using Input-Output Examples. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Austin, Texas, USA) *(POPL '11)*. Association for Computing Machinery, New York, NY, USA, 317–330. https://doi.org/10.1145/1926385.1926423

Sumit Gulwani, Alex Polozov, and Rishabh Singh. 2017. *Program Synthesis*. Vol. 4. NOW. 1–119 pages. https://www.microsoft.com/en-us/research/publication/program-synthesis/

David Gunning and David Aha. 2019. DARPA's Explainable Artificial Intelligence (XAI) Program. *AI Magazine* 40, 2 (Jun. 2019), 44–58. https://doi.org/10.1609/aimag.v40i2.2850

Will Hamilton, Zhitao Ying, and Jure Leskovec. 2017. Inductive Representation Learning on Large Graphs. In *Advances in Neural Information Processing Systems*, I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett (Eds.), Vol. 30. Curran Associates, Inc. https://proceedings.neurips.cc/paper/2017/file/5dd9db5e033da9c6fb5ba83c7a7ebea9-Paper.pdf

Qiang Huang, Makoto Yamada, Yuan Tian, Dinesh Singh, and Yi Chang. 2022. GraphLIME: Local Interpretable Model Explanations for Graph Neural Networks. *IEEE Transactions on Knowledge and Data Engineering* (2022), 1–6. https://doi.org/10.1109/TKDE.2022.3187455

Akihiro Inokuchi, Takashi Washio, and Hiroshi Motoda. 2000. An Apriori-Based Algorithm for Mining Frequent Substructures from Graph Data. In *Proceedings of the 4th European Conference on Principles of Data Mining and Knowledge Discovery (PKDD '00)*. Springer-Verlag, Berlin, Heidelberg, 13–23.

Minseok Jeon. 2024. PL4XGL: A Programming Language Approach to Explainable Graph Learning (Artifact). (2024). https://doi.org/10.5281/zenodo.10783891

Jaykumar Kakkad, Jaspal Jannu, Kartik Sharma, Charu Aggarwal, and Sourav Medya. 2023. A Survey on Explainability of Graph Neural Networks. arXiv:2306.01958 [cs.LG]

Thomas N. Kipf and Max Welling. 2017. Semi-Supervised Classification with Graph Convolutional Networks. In *International Conference on Learning Representations (ICLR)*.

Yuquan Li, Chang-Yu Hsieh, Ruiqiang Lu, Xiaoqing Gong, Xiaorui Wang, Pengyong Li, Shuo Liu, Yanan Tian, Dejun Jiang, Jiaxian Yan, Qifeng Bai, Huanxiang Liu, Shengyu Zhang, and Xiaojun Yao. 2022. An adaptive graph learning method for automated molecular interactions and properties predictions. *Nature Machine Intelligence* 4, 7 (2022), 645–651. https://doi.org/10.1038/s42256-022-00501-8

Meng Liu, Youzhi Luo, Limei Wang, Yaochen Xie, Hao Yuan, Shurui Gui, Zhao Xu, Haiyang Yu, Jingtun Zhang, Yi Liu, Keqiang Yan, Bora Oztekin, Haoran Liu, Xuan Zhang, Cong Fu, and Shuiwang Ji. 2021. DIG: A Turnkey Library for Diving into Graph Deep Learning Research. *arXiv preprint arXiv:2103.12608* (2021).

Yunchao "Lance" Liu, Yu Wang, Oanh Vu, Rocco Moretti, Bobby Bodenheimer, Jens Meiler, and Tyler Derr. 2022. Interpretable Chirality-Aware Graph Neural Network for Quantitative Structure Activity Relationship Modeling in Drug Discovery. *bioRxiv* (2022). https://doi.org/10.1101/2022.08.24.505155 arXiv:https://www.biorxiv.org/content/early/2022/08/26/2022.08.24.505155.full.pdf

Ana Lucic, Maartje A. Ter Hoeve, Gabriele Tolomei, Maarten De Rijke, and Fabrizio Silvestri. 2022. CF-GNNExplainer: Counterfactual Explanations for Graph Neural Networks. In *Proceedings of The 25th International Conference on Artificial Intelligence and Statistics (Proceedings of Machine Learning Research, Vol. 151)*, Gustau Camps-Valls, Francisco J. R. Ruiz, and Isabel Valera (Eds.). PMLR, 4499–4511. https://proceedings.mlr.press/v151/lucic22a.html

Dongsheng Luo, Wei Cheng, Dongkuan Xu, Wenchao Yu, Bo Zong, Haifeng Chen, and Xiang Zhang. 2020. Parameterized Explainer for Graph Neural Network. In *Proceedings of the 34th International Conference on Neural Information Processing Systems* (Vancouver, BC, Canada) *(NIPS'20)*. Curran Associates Inc., Red Hook, NY, USA, Article 1646, 12 pages.

Anders Miltner, Adrian Trejo Nuñez, Ana Brendel, Swarat Chaudhuri, and Isil Dillig. 2022. Bottom-up Synthesis of Recursive Functional Programs Using Angelic Execution. *Proc. ACM Program. Lang.* 6, POPL, Article 21 (jan 2022), 29 pages. https://doi.org/10.1145/3498682

Yoichi Morofuji and Shinsuke Nakagawa. 2020. Drug Development for Central Nervous System Diseases Using In vitro Blood-brain Barrier Models and Drug Repositioning. *Curr Pharm Des* 26, 13 (2020), 1466–1485. https://doi.org/10.2174/1381612826666200224112534

Jinyoung Park, Sungdong Yoo, Jihwan Park, and Hyunwoo J Kim. 2022. Deformable Graph Convolutional Networks. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 36. 7949–7956.

Hongbin Pei, Bingzhe Wei, Kevin Chen-Chuan Chang, Yu Lei, and Bo Yang. 2020. Geom-GCN: Geometric Graph Convolutional Networks. In *International Conference on Learning Representations*. https://openreview.net/forum?id=S1e2agrFvS

Phillip E. Pope, Soheil Kolouri, Mohammad Rostami, Charles E. Martin, and Heiko Hoffmann. 2019. Explainability Methods for Graph Convolutional Neural Networks. In *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. 10764–10773. https://doi.org/10.1109/CVPR.2019.01103

T. Ramraj and R. Prabhakar. 2015. Frequent Subgraph Mining Algorithms – A Survey. *Procedia Computer Science* 47 (2015), 197–204. https://doi.org/10.1016/j.procs.2015.03.198 Graph Algorithms, High Performance Implementations and Its Applications (ICGHIA 2014).

Susie Xi Rao, Shuai Zhang, Zhichao Han, Zitao Zhang, Wei Min, Zhiyao Chen, Yinan Shan, Yang Zhao, and Ce Zhang. 2021. XFraud: Explainable Fraud Transaction Detection. *Proc. VLDB Endow.* 15, 3 (nov 2021), 427–436. https://doi.org/10.14778/3494124.3494128

Michael Sejr Schlichtkrull, Nicola De Cao, and Ivan Titov. 2021. Interpreting Graph Neural Networks for NLP With Differentiable Edge Masking. In *International Conference on Learning Representations*. https://openreview.net/forum?id=WznmQa42ZAx

Thomas Schnake, Oliver Eberle, Jonas Lederer, Shinichi Nakajima, Kristof T Schutt, Klaus-Robert Mueller, and Gregoire Montavon. 2021. Higher-Order Explanations of Graph Neural Networks via Relevant Walks. *IEEE Transactions on Pattern Analysis and Machine Intelligence* (2021), 1–1. https://doi.org/10.1109/tpami.2021.3115452

Mengying Sun, Sendong Zhao, Coryandar Gilvary, Olivier Elemento, Jiayu Zhou, and Fei Wang. 2019. Graph convolutional networks for computational drug development and discovery. *Briefings in Bioinformatics* 21, 3 (06 2019), 919–935. https://doi.org/10.1093/bib/bbz042 arXiv:https://academic.oup.com/bib/article-pdf/21/3/919/33227266/bbz042.pdf

Abhishek Udupa, Arun Raghavan, Jyotirmoy V. Deshmukh, Sela Mador-Haim, Milo M.K. Martin, and Rajeev Alur. 2013. TRANSIT: Specifying Protocols with Concolic Snippets. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Seattle, Washington, USA) *(PLDI '13)*. Association for Computing Machinery, New York, NY, USA, 287–296. https://doi.org/10.1145/2491956.2462174

Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. 2018. Graph Attention Networks. In *International Conference on Learning Representations*.

Minh Vu and My T. Thai. 2020a. PGM-Explainer: Probabilistic Graphical Model Explanations for Graph Neural Networks. In *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin (Eds.), Vol. 33. Curran Associates, Inc., 12225–12235. https://proceedings.neurips.cc/paper/2020/file/8fb134f258b1f7865a6ab2d935a897c9-Paper.pdf

Minh N. Vu and My T. Thai. 2020b. PGM-Explainer: Probabilistic Graphical Model Explanations for Graph Neural Networks. In *Proceedings of the 34th International Conference on Neural Information Processing Systems* (Vancouver, BC, Canada) *(NIPS'20)*. Curran Associates Inc., Red Hook, NY, USA, Article 1025, 11 pages.

Yingxin Wu, Xiang Wang, An Zhang, Xiangnan He, and Tat-Seng Chua. 2022. Discovering Invariant Rationales for Graph Neural Networks. In *International Conference on Learning Representations*. https://openreview.net/forum?id=hGXij5rfiHw

Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and Philip S. Yu. 2021. A Comprehensive Survey on Graph Neural Networks. *IEEE Transactions on Neural Networks and Learning Systems* 32, 1 (2021), 4–24. https://doi.org/10.1109/TNNLS.2020.2978386

Zhenqin Wu, Bharath Ramsundar, Evan N. Feinberg, Joseph Gomes, Caleb Geniesse, Aneesh S. Pappu, Karl Leswing, and Vijay Pande. 2018. MoleculeNet: a benchmark for molecular machine learning. *Chem. Sci.* 9 (2018), 513–530. Issue 2. https://doi.org/10.1039/C7SC02664A

Jiacheng Xiong, Zhaoping Xiong, Kaixian Chen, Hualiang Jiang, and Mingyue Zheng. 2021. Graph neural networks for automated de novo drug design. *Drug Discovery Today* 26, 6 (2021), 1382–1393. https://doi.org/10.1016/j.drudis.2021.02.011

Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. 2019. How Powerful are Graph Neural Networks?. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net. https://openreview.net/forum?id=ryGs6iA5Km

Keyulu Xu, Chengtao Li, Yonglong Tian, Tomohiro Sonobe, Ken-ichi Kawarabayashi, and Stefanie Jegelka. 2018. Representation Learning on Graphs with Jumping Knowledge Networks. In *Proceedings of the 35th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 80)*, Jennifer Dy and Andreas Krause (Eds.). PMLR, 5453–5462. https://proceedings.mlr.press/v80/xu18c.html

Xifeng Yan and Jiawei Han. 2002. gSpan: graph-based substructure pattern mining. In *2002 IEEE International Conference on Data Mining, 2002. Proceedings.* 721–724. https://doi.org/10.1109/ICDM.2002.1184038

Ziyuan Ye, Rihan Huang, Qilin Wu, and Quanying Liu. 2023. SAME: Uncovering GNN Black Box with Structure-aware Shapley-based Multipiece Explanations. In *Thirty-seventh Conference on Neural Information Processing Systems*. https://openreview.net/forum?id=kBBsj9KRgh

Zhitao Ying, Dylan Bourgeois, Jiaxuan You, Marinka Zitnik, and Jure Leskovec. 2019. GNNExplainer: Generating Explanations for Graph Neural Networks. In *Advances in Neural Information Processing Systems*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett (Eds.), Vol. 32. Curran Associates, Inc. https://proceedings.neurips.cc/paper_files/paper/2019/file/d80b7040b773199015de6d3b4293c8ff-Paper.pdf

Hao Yuan, Jiliang Tang, Xia Hu, and Shuiwang Ji. 2020. XGNN: Towards Model-Level Explanations of Graph Neural Networks. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining* (Virtual Event, CA, USA) *(KDD '20)*. Association for Computing Machinery, New York, NY, USA, 430–438. https:

//doi.org/10.1145/3394486.3403085

Hao Yuan, Haiyang Yu, Shurui Gui, and Shuiwang Ji. 2022. Explainability in Graph Neural Networks: A Taxonomic Survey. *IEEE Transactions on Pattern Analysis and Machine Intelligence* (2022), 1–19. https://doi.org/10.1109/TPAMI.2022.3204236

Hao Yuan, Haiyang Yu, Jie Wang, Kang Li, and Shuiwang Ji. 2021. On Explainability of Graph Neural Networks via Subgraph Explorations. In *Proceedings of the 38th International Conference on Machine Learning (ICML)*. 12241–12252.

Yue Zhang, David Defazio, and Arti Ramesh. 2021. RelEx: A Model-Agnostic Relational Model Explainer. In *Proceedings of the 2021 AAAI/ACM Conference on AI, Ethics, and Society* (Virtual Event, USA) *(AIES '21)*. Association for Computing Machinery, New York, NY, USA, 1042–1049. https://doi.org/10.1145/3461702.3462562

Zaixi Zhang, Qi Liu, Hao Wang, Chengqiang Lu, and Cheekong Lee. 2022. ProtGNN: Towards Self-Explaining Graph Neural Networks. *Proceedings of the AAAI Conference on Artificial Intelligence* 36, 8 (Jun. 2022), 9127–9135. https://doi.org/10.1609/aaai.v36i8.20898

Marinka Zitnik, Monica Agrawal, and Jure Leskovec. 2018. Modeling Polypharmacy Side Effects with Graph Convolutional Networks. *bioRxiv* (2018). https://doi.org/10.1101/258814 arXiv:https://www.biorxiv.org/content/early/2018/05/18/258814.full.pdf

Received 2023-11-16; accepted 2024-03-31