Global and Local Explanations for Negative GNN Predictions

Kehan Pang[1], Wenfei Fan[1,2,3], Min Xie[2], Dandan Lin[2]
[1]Beihang University [2]Shenzhen Institute of Computing Sciences [3]University of Edinburgh
pangkehan@buaa.edu.cn,wenfei@inf.ed.ac.uk,{xiemin,lindandan}@sics.ac.cn

ABSTRACT

This paper studies explanations for graph neural network (GNN) classifiers $\mathcal{M}$ when $\mathcal{M}$ makes a negative prediction, such as loan denials, paper rejections, or declined job applications. The objective is to both (a) globally explain the general behavior of $\mathcal{M}$; and (b) suggest *counterfactual explanations* locally at a vertex u in a graph, which are necessary changes to features/topology around u for $\mathcal{M}$ to swap its prediction at u. We propose a class of rules which treat negative $\mathcal{M}$-predictions as their consequences. We develop algorithms to (a) learn such rules as global explanations and (b) compute local counterfactual explanations, by applying the learned rules. Over real-life graphs, our algorithms are on average 70.16% and 189.68% higher in recognizability and reliability than prior global methods, and 33.60% and 3.25× better than previous local (counterfactual) methods in fidelity and sparsity, respectively.

1 INTRODUCTION

With the prevalent use of machine learning (ML) models comes the need for explaining ML predictions. This is to provide users with insights and establish their trust in the predictions, and help developers debug ML models by revealing errors or bias in training data. Moreover, explanations have become a must for the outcomes of automated decision algorithms by the EU General Data Protection Regulation [87]. The need for explanation is particularly evident when the models produce negative outcomes, when the users often seek to understand, *e.g.,* why their loan was denied, why their paper got rejected, and why their job application was declined.

Consider explaining bad predictions of graph neural networks (GNNs), which have been used in fraud detection [61, 74] and drug discovery [58, 71, 79, 91], among other things. To explain GNNs, ML models [10, 17, 59, 63, 80, 85] have been trained to identify subgraphs and features on which the prediction is made, and compute *counterfactual explanations* [68], *i.e.,* necessary graph perturbations to swap the prediction. One can adopt (a) *local explanations* that given a bad prediction at an *input instance*, suggest perturbations for a specific user to swap the prediction, or (b) *global explanations* that reveal *input-independent* behavior of the model. Although each explanation offers a unique perspective, most prior methods primarily focus on one of them. Few methods [5, 16] consider both, but they either exhibit poor efficiency or rely on simple heuristic.

This paper develops a logical approach to provide both global and local explanations in a unified framework that overcomes the limitations of existing studies. Its core is a class of logic rules, referred to RxGNNs (Rules for eXplaining GNN classifiers). Given a GNN classifier $\mathcal{M}$ and a vertex variable x_0 at which $\mathcal{M}$ predicts false, an RxGNNs has the form of $Q[\bar{x}, x_0](X \to \neg\mathcal{M}(x_0))$, where Q is a graph pattern, and X is a collection of predicates. Intuitively, Q identifies topological structures around x_0 relevant to the decision, and X discloses conditions on vertex features. These rules provide *global explanations* to reveal what structures and features are most responsible for predictions of $\mathcal{M}$. When x_0 is instantiated to a particular vertex u in a graph G, the RxGNNs provide *local explanations* at u.

Example 1: Consider a graph G in Figure 1(a) of employment network (simplified at the bottom-right). A GNN model $\mathcal{M}_1$ suggests to decline the job application of a person u_0, denoted by $\neg\mathcal{M}_1(u_0)$.

To explain the behavior of $\mathcal{M}_1$, we discover a set of RxGNNs pertaining to the negative predictions of $\mathcal{M}_1$ as its global explanations. An example RxGNN is shown at the top-right of Figure 1(b):

$$\varphi_1 = Q_1[\bar{x}, x_0](X_1 \to \neg\mathcal{M}_1(x_0)),$$

where X_1 is $x_0.\#\text{experience} < 3$ years $\wedge x_1.\text{headcount} = 1 \wedge x_3.\text{QS\_ranking} > 500 \wedge x_4.\text{GPA} > x_0.\text{GPA} \wedge \neg 1\text{WL}(x_4) \wedge x_0.\text{id} \neq x_5.\text{id} \wedge x_0.\text{id} \neq x_6.\text{id} \wedge x_5.\text{id} \neq x_6.\text{id}$. Here $\neg 1\text{WL}(x_4)$ is a (negated) 1-WL predicate (1-dimensional Weisfeiler Leman (1-WL) test, a graph theoretic technique widely used for comparing the structure of graphs, to be defined in Section 2.1), indicating that x_4 should be predicted false by $\mathcal{M}_1$. Pattern Q_1 together with conditions X_1 of φ_1 explain why $\mathcal{M}_1$ suggests to decline the application of x_0, *because* (a) x_0 is not experienced enough (#experience < 3 years), (b) the school x_3 where x_0 got his highest degree is not top-tier, with QS ranking below 500, (c) the post x_1 that x_0 is applying for has only 1 headcount but it has been applied by at least 3 distinct applicants (*i.e.,* x_0, x_5 and x_6 whose user IDs are different), and (d) another person x_4, who has the same major as x_0 and obtains degree from the same school as x_0, has a higher GPA than x_0, but x_4 is already denied by 1-WL test.

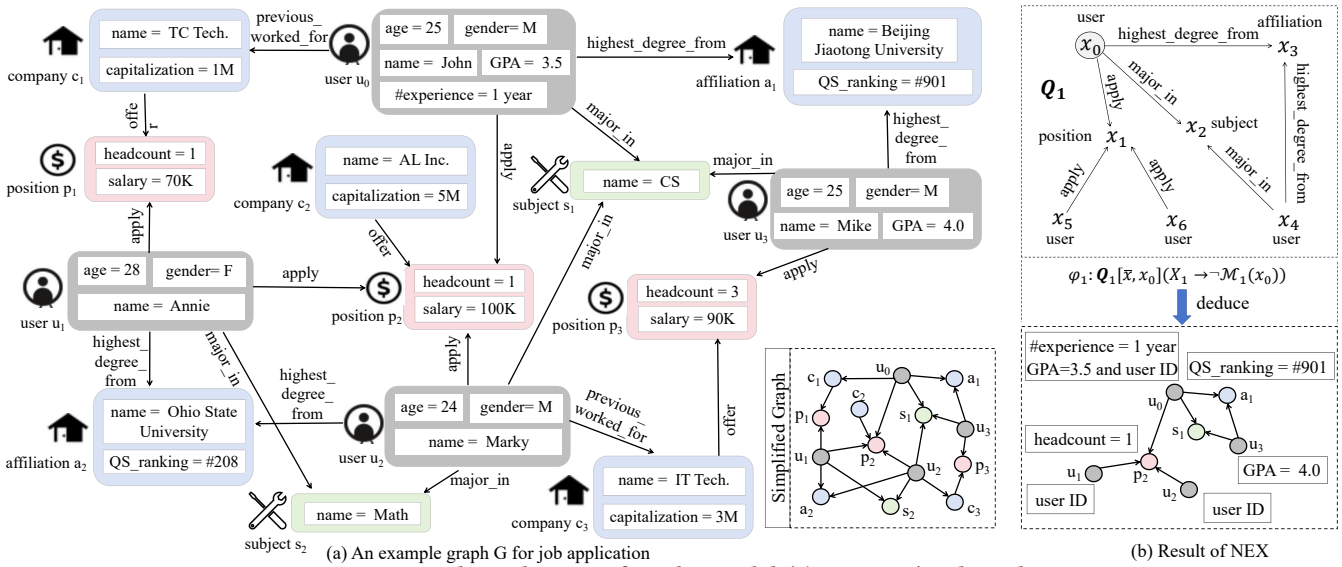
When x_0 is instantiated to person u_0 (John) in G (Figure 1(b)), φ_1 provides a local explanation: the application of John is denied by $\mathcal{M}_1$, since (a) John only has 1 year of experience and the school from which he graduated has QS-ranking #901, (b) Mike, his schoolmate, is also majored in CS and has a higher GPA, but Mike's application is rejected, and (c) the post that John is applying for has only 1 headcount but is applied by 3 distinct persons (John, Annie and Marky).

To swap the prediction of $\mathcal{M}_1$ at u_0 (*i.e., counterfactual explanation*), prior methods may suggest unreasonable changes (*e.g.,* change the major he has already studied). In contrast, our method advocates feasible suggestions, *e.g.,* accumulate more working experiences or get a better degree. A real-life case study is reported in Section 6. □

Developing both global and local explanations is hard since it requires (a) extracting patterns to capture $\mathcal{M}_1$'s behavior, (b) supporting diverse predicates (constant, variable, 1-WL) to ensure enough expressivity, (c) applying efficient updates for prediction switches.

Contributions & organization. We propose to uniformly provide both global and local explanations for GNN-based classifier $\mathcal{M}$.

(1) Rules for eXplaining GNN (RxGNNs) (Section 2). We propose a class of rules, RxGNNs, which extend prior graph rules by (a) supporting general predicates, including negation, over *general* graph patterns; (b) treating negative $\mathcal{M}$ predictions as consequences; and (c) including 1-WL test [65, 89], which is at least as expressive as GNN classifiers [13, 40, 41, 103]; thus, RxGNNs can fully explain $\mathcal{M}$.



(a) An example graph G for job application (b) Result of NEX

Figure 1: An example explanation for why model $\mathcal{M}$ rejects u_0's job application

(2) NEX*: Negative prediction EXplanations* (Section 3). We develop a uniform explanation framework, named NEX. Given a GNN classifier $\mathcal{M}$, NEX first discovers a set Σ of RxGNNs pertaining to the negative predictions of $\mathcal{M}$ as global explanation. Moreover, when $\mathcal{M}$ predicts negative at a particular vertex u in a graph G, NEX provides a counterfactual explanation as a minimum set of graph perturbations such that no RxGNNs in Σ can be applied at u, *i.e.,* necessary changes for the user to make for swapping the $\mathcal{M}$ decision.

(3) Model-guided discovery (Section 4). We develop an algorithm for discovering RxGNNs guided by model $\mathcal{M}$. To provide faithful explanation, we identify fundamental sub-structures, called *motifs*, that are mostly responsible for $\mathcal{M}$'s predictions. These motifs are strategically composed into graph patterns. We then generate preconditions for these patterns, by enforcing a novel predicate ordering for effective processing. The algorithm is parallelly scalable [56], *i.e.,* it can reduce parallel runtime when given more processors.

(4) Counterfactual explanations (Section 5). For a specific vertex u in graph G with negative outcome $\mathcal{M}(u)$, we want a set of graph perturbations that makes no RxGNNs in Σ applicable at u anymore. It suggests necessary changes for the user to make in order to swap $\mathcal{M}(u)$ from negative to positive. While the problem is NP-hard, we develop a cost-effective strategy to decide how to remove edges or modify vertex features, minimizing the perturbations to G.

(5) Experimental study (Section 6). Using four real-life graphs, we find: (a) The RxGNNs discovered by NEX provide accurate global explanations. Their recognizability and reliability are 70.16% and 189.68% higher than the baselines on average, respectively. (b) NEX gives effective counterfactual explanations. Its average fidelity and sparsity are 33.60% and 3.25× better than the prior methods, respectively. (c) NEX is efficient, *e.g.,* its discovery algorithm is parallelly scalable, and it only takes 1.38s on average to generate a counterfactual explanation on a graph with 15.4M vertices and 12.7M edges.

We discuss related work in Section 7 and future work in Section 8.

2 RULES FOR EXPLAINING GNN OUTCOMES

This section first reviews 1-WL test (Section 2.1), and then defines rules for explaining negative GNN classifications (Section 2.2).

2.1 Graph Patterns and 1-WL Test

Assume two countably infinite sets of symbols, denoted by Λ and Υ, for edge/vertex labels and vertex attributes, respectively.

Graphs. We consider directed labeled graphs $G = (V, E, L_G, F_A)$, where (a) V is a finite set of vertices; (b) $E \subseteq V \times \Lambda \times V$ is a finite set of edges, in which $e = (v, l, v')$ denotes an edge labeled with $l \in \Lambda$ from vertex v to v'; (c) each vertex $v \in V$ carries a label $L_G(v) \in \Lambda$, and a tuple $F_A(v) = (A_1 = a_1, \ldots, A_n = a_n)$ of *attributes* of a finite arity, where $A_i \in \Upsilon$ and a_i is a constant, written as $v.A_i = a_i$ ($A_i \neq A_j$ if $i \neq j$), for features of v. Different vertices may carry different attributes, while edges only carry labels and no attributes.

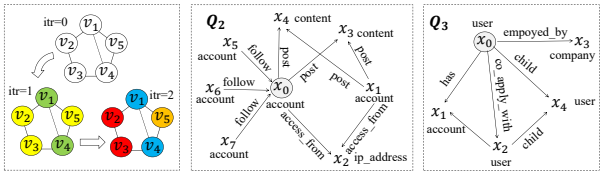
Patterns. A (connected) *graph pattern* is $Q[\bar{x}, x_0] = (V_Q, E_Q, L_Q)$, where (1) V_Q (resp. E_Q) is a finite set of *pattern vertices* (resp. *edges*); (2) L_Q assigns a label $L_Q(v) \in \Lambda$ to each vertex $v \in V_Q$; (3) $\bar{x}$ is a list of distinct variables, where each $x \in \bar{x}$ denotes a vertex $v \in V_Q$ [30, 35], and (4) x_0 is a designated vertex in $\bar{x}$ for an entity of interest.

Pattern matching. Following [6, 32], a *match* of pattern $Q[\bar{x}, x_0]$ in graph G is defined as a homomorphism h from Q to G such that (a) for each pattern vertex $v \in V_Q$, $L_Q(v) = L_G(h(v))$; and (b) for each pattern edge $e = (v, l, v')$ in Q, $e' = (h(v), l, h(v'))$ is an edge in G.

GNN. Graph neural networks (GNNs) [44, 81, 84, 100] utilize the graph topology and the features of vertices to learn a representation embedding for each vertex, which can subsequently be applied to vertex classification. Let $\mathbf{e}_v$ be the initial feature vector of vertex v, derived from its attributes. An L-layer GNN iteratively refines the embedding of each vertex by aggregating the embeddings of its L-hop neighbors. The embedding $\mathbf{e}_v^t$ for v at the t-th layer is:

$$\mathbf{e}_v^t = \text{Update}(\mathbf{e}_v^{t-1}, \text{Aggregate}(\{\mathbf{e}_{v'}^{t-1}\} \mid v' \in \mathcal{N}(v))),$$

where Aggregate and Update are aggregation and update functions, respectively, and $\mathcal{N}(v)$ is the set of 1-hop neighbors of v. After L iterations, the model outputs the final embedding $\mathbf{e}_v^L$ for v and the probability of its classification label is predicted as $f(\mathbf{e}_v^L)$. Here $f(\mathbf{e}_v^L)$ is a function learned *w.r.t.* the classes in the training graph. To simplify, we consider binary vertex classification following [14, 81], *i.e.,* given a predefined threshold λ for the prediction strength, $\mathcal{M}(v)$ returns true (resp. false) for a vertex v if $f(\mathbf{e}_v^L) \geq \lambda$ (resp. $f(\mathbf{e}_v^L) < \lambda$).



(a) An example of 1-WL test (b) Examples of graph patterns

Figure 2: Running examples

Example 2: As shown in Figure 1(b), the graph pattern Q_1 depicts entities and relationships in job applications (*e.g.,* users, companies, positions and apply) for making the false prediction in Example 1. □

1-WL test. We review 1-dimensional Weisfeiler Leman (1-WL) test [89]. Given graph G where each vertex is initialized with a color (decided by its initial feature vector), 1-WL test works via color refinement (cf. [40]): for all colors c in current coloring and all vertices v and v' of color c, v and v' get different colors in new coloring if there is some color d such that v and v' have different numbers of neighbors of color d. This refinement iterates until no more changes can be made. Vertices with the same color are put into the same class.

Example 3: Consider Figure 2(a) (an example borrowed from [103]). Initially, all vertices are white. The coloring is refined iteratively. In the first iteration, v_1 and v_5 are marked with different colors since they have 3 and 2 white children, respectively. After two iterations, no more changes can be made and the final coloring is shown; v_2 and v_3 have the same color and are put into the same class. □

Notations. We will use the following notations (see Table 1).

(1) We use $\neg\mathcal{M}(x)$ to denote a negative prediction (false) of model $\mathcal{M}$ at a vertex in a graph (*i.e.,* $f(\mathbf{e}_v^L) < \lambda$ for a predefined threshold λ). We will treat $\neg\mathcal{M}(x)$ as a predicate, returning a Boolean value.

(2) We use $\neg 1\mathsf{WL}(x)$ to denote that there exists a vertex y such that $\mathcal{M}(y)$ is false, and x and y are similar by 1-WL test, *i.e.,* x should be put in the same class as y. We also treat $\neg 1\mathsf{WL}(x)$ as a predicate. The 1-WL test can be computed in $O((|V|+|E|)\log|V|)$ time (cf. [40]).

2.2 Rules with Negated Predicates

We next define RxGNNs, Rules for eXplaining GNN classifiers.

Predicates. We define a *logic predicate* of a pattern $Q[\bar{x}, x_0]$ as:

$$p ::= x.A \oplus y.B \mid x.A \oplus c \mid \neg 1\mathsf{WL}(x),$$

where $\oplus$ is one of $=, \neq, <, \leq, >, \geq$; x and y are variables in $\bar{x}$; c is a constant; A and B are attributes; and $\neg 1\mathsf{WL}(x)$ is the *1-WL predicate*.

RxGNNs. Given a GNN classifier $\mathcal{M}$, a *Rule φ for eXplaining $\mathcal{M}$* is

$$Q[\bar{x}, x_0](X \rightarrow \neg\mathcal{M}(x_0)),$$

where $Q[\bar{x}, x_0]$ is a graph pattern, and X is a conjunction of predicates of $Q[\bar{x}, x_0]$. We refer to Q and $X \rightarrow \neg\mathcal{M}(x_0)$ as the *pattern* and *dependency*, and X and $\neg\mathcal{M}(x_0)$ as the *precondition* and *consequence* of φ, respectively. We consider RxGNNs that take the same classifier $\mathcal{M}$ in their consequence, *i.e., RxGNNs pertaining to model $\mathcal{M}$*.

One can plug in an arbitrary GNN classifier $\mathcal{M}$ in RxGNNs, *e.g.,* GCN [55], GAT [84], and GIN [92], as long as $\mathcal{M}$ is *pre-trained, fixed* (*i.e.,* its architecture and weights no longer change) and *deterministic* (*i.e.,* it generates the same result for the same input) [16, 69, 70, 72].

Intuitively, pattern $Q[\bar{x}, x_0]$ identifies topological structures relevant to entity x_0 that are inspected by $\mathcal{M}$; precondition X catches the interactions of vertex features. Together Q and X explain why $\mathcal{M}(x_0)$ makes its prediction. In particular, $\neg 1\mathsf{WL}(x)$ explains how GNN classifies entities in terms of 1-WL test. We separate Q and X in order to (a) visualize the topology of entities by Q, and (b) speed up evaluation by leveraging the locality of pattern matching.

Notations	Definitions
$G, Q[\bar{x}, x_0]$	graph, graph pattern
$\neg\mathcal{M}(x_0)$	an L-layer GNN model that predicts false at a vertex x_0
$\neg 1\mathsf{WL}(x)$	the (negated) predicate for 1-WL test
p, X	a predicate, the precondition (a conjunction of logic predicates)
φ, Σ	an RxGNN $\varphi = Q[\bar{x}, x_0](X \rightarrow \neg\mathcal{M}(x_0))$, a set of RxGNNs
σ, δ, N	thresholds of support and confidence, max pattern size
$\mathsf{supp}(\,), \mathsf{conf}(\,)$	the support and confidence of an RxGNN φ on graph G
$\mathbb{O}, G \ominus \mathbb{O}$	a set of perturbations in G, the graph G perturbed by $\mathbb{O}$
$\mathsf{cost}(v), \mathsf{eff}(v)$	the cost and effectiveness of a vertex v

Table 1: Notations

Example 4: Below are RxGNNs with patterns in Figure 2(b).

(1) An RxGNN φ_1 is given in Example 1 to explain why $\mathcal{M}_1$ predicts that the job application of x_0 is declined. It extracts important features from x_0's background and the degree of market competition.

(2) $\varphi_2 = Q_2[\bar{x}, x_0](X_2 \rightarrow \neg\mathcal{M}_2(x_0))$, where X_2 is $\neg 1\mathsf{WL}(x_1) \wedge x_3.\mathsf{content} = x_4.\mathsf{content} \wedge x_5.\mathsf{account\_age} < 24\mathsf{h} \wedge x_6.\mathsf{account\_age} < 24\mathsf{h} \wedge x_7.\mathsf{account\_age} < 24\mathsf{h}$. It says that $\mathcal{M}_2$ flags a user account x_0 for suspension since (a) x_0 behaves like a suspended account x_1 (which is in the negative class by 1-WL test), *i.e.,* both post repetitive contents (*e.g.,* bot-generated spammy "Great post!"), and they have been accessed from the same IP address; and (b) many followers of x_0 are created within 24 hours (bot-like accounts).

(3) $\varphi_3 = Q_3[\bar{x}, x_0](X_3 \rightarrow \neg\mathcal{M}_3(x_0))$, where X_3 is $x_0.\mathsf{debt\_to\_income} \geq 40\% \wedge x_3.\mathsf{default\_history} = \mathsf{yes} \wedge \neg 1\mathsf{WL}(x_4) \wedge x_2.\mathsf{credit} \leq x_4.\mathsf{credit} \wedge x_1.\mathsf{balance} < 10\mathsf{K}$. It says that $\mathcal{M}_3$ suggests to deny the loan of x_0 because (a) x_0 has high debt-to-income ratio ($\geq 40\%$) and x_0 is employed by a company defaulted on a loan, (b) an immediate relative x_4 of x_0 is put in the negative class by 1-WL test, (c) the co-applicant (spouse) x_2 of x_0 has even worse credit than x_4 (who is in negative class) and their joint account has low balance ($< 10\mathsf{K}$). □

RxGNNs extend prior graph rules (*e.g.,* GEDs [32], TACOs [31], GARs [30], REPs [23], TIEs [22] and GCRs [24]) with properties essential for expressive counterfactual explanations (see [4] for more):

(1) A full range of predicates. RxGNNs not only allow constant, variable and ML predicates, they also support (a) limited negation, *i.e.,* $\neg\mathcal{M}(x_0)$, $\neg 1\mathsf{WL}(x)$ and $\neg(x.A \oplus y.B)$ by the dual of $\oplus$ (*e.g.,* $\geq$ vs. $<$, and $=$ vs. $\neq$), which provides added modeling capabilities that are essential for flipping predictions; and (b) 1-WL test as predicate, whose computation explicitly considers message passing via color refinement. Indeed, 1-WL is at least as expressive as GNN classifiers [13, 40, 41, 103] and thus, RxGNNs can explain the predictions of $\mathcal{M}$.

(2) General patterns. RxGNNs are defined on general patterns, beyond restricted star patterns in [22–24]. Note that general patterns are more expressive for capturing complex topological structures.

(3) Negative prediction as consequences. RxGNNs take $\neg\mathcal{M}(x_0)$ as consequences, enabling counterfactual explanations via updates.

Semantics. Consider a graph G and an RxGNN $\varphi = Q[\bar{x}, x_0](X \rightarrow \neg\mathcal{M}(x_0))$. Denote by h a match in the pattern Q of φ in graph G.

We say that match h *satisfies* a predicate p of $Q[\bar{x}, x_0]$, denoted by $h \models p$, if the following conditions are satisfied: (a) when p is $x.A \oplus y.B$, the vertex $h(x)$ (resp. $h(y)$) carries attribute A (resp. B), and $h(x).A \oplus h(y).B$; similarly for $x.A \oplus c$, (b) when p is $\neg 1\mathsf{WL}(x)$, the 1-WL test puts $h(x)$ in the same class as a vertex v at which $\mathcal{M}(v)$ is false; and (c) for $\neg\mathcal{M}(x_0)$, $\mathcal{M}$ predicts negative at $h(x_0)$.

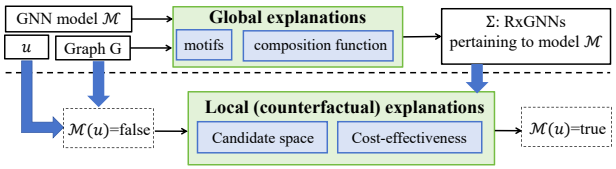


Figure 3: The framework of NEX

We write $h \models X$ if h satisfies *all* predicates in X. We write $h \models \varphi$ if $h \models X$ entails $h \models \neg\mathcal{M}(x_0)$. For a vertex u in G, we refer to mapping h as a *witness of φ pivoted at u* if $h \models \varphi$ and $h(x_0) = u$. We say that RxGNN φ is *applicable at u* if there exists a witness pivoted at u. We say that a graph G *satisfies* φ, denoted by $G \models \varphi$, if for all matches h of $Q[\bar{x}, x_0]$ in G such that if $h \models X$, then $h \models \varphi$. We write $G \models \Sigma$ for a set Σ of RxGNNs if for all $\varphi \in \Sigma$, $G \models \varphi$.

3 GLOBAL AND LOCAL EXPLANATIONS

This section presents an overview of NEX, which provides both global and local explanations with RxGNNs, as outlined in Figure 3.

Global explanations. Given a GNN $\mathcal{M}$ and a graph G, we discover a set Σ of RxGNNs pertaining to the negative predictions of $\mathcal{M}$, where each RxGNN is composed from a number of important substructures, called *motifs*, from graph G. As evidenced in [80], certain sub-structures play an essential role in the GNN's prediction, *e.g.,* the Nitrobenzene structure is likely a key indicator of mutagenicity. We use such prediction signals from $\mathcal{M}$ to guide the discovery so that the rules in Σ explain the instance-independent behaviors of $\mathcal{M}$, cover different cases of $\mathcal{M}$ via 1-WL test, and present the most discriminative structures and features for the predictions. Therefore, the set Σ of RxGNNs can serve as global explanations of $\mathcal{M}$.

Local explanations. We provide a counterfactual explanation with Σ for each vertex u in graph G where $\mathcal{M}(u) = \text{false}$. As all RxGNNs in Σ take $\neg\mathcal{M}(x_0)$ as consequences, they are non-conflicting. Moreover, rule discovery is a one-time offline process, allowing Σ to be reused for local explanations across all inputs u with $\mathcal{M}(u) = \text{false}$.

Counterfactual explanations. Consider a witness h of φ that maps variable x to v in G. A *perturbation* in G is either (a) *feature modification* by changing an attribute value of v such that a predicate $p \in X$ involving x is no longer satisfied, or (b) *edge removal* by deleting an incident edge of v such that h no longer matches the pattern Q of φ.

A *counterfactual explanation* for $\neg\mathcal{M}(u)$ is the minimum set $\mathbb{O}$ of perturbations in G such that no $\varphi \in \Sigma$ is applicable at u in the updated G, denoted by $G \ominus \mathbb{O}$. Intuitively, $\mathbb{O}$ consists of *necessary changes* to swap the prediction $\mathcal{M}(u)$ from negative to positive.

Example 5: In Example 4, φ_3 can serve as part of a global explanation of unsuccessful loan applications, by considering employment ties (*e.g.,* $(x_0, \text{employed\_by}, x_3)$), credit history (*e.g.,* x_2.credit) and social connections (*e.g.,* (x_0, child, x_4)) of applicants. Consider a witness h_3 of φ_3 at u. To swap the prediction $\mathcal{M}_3(u)$ from negative to positive, u can *e.g.,* pay off credit card debt and deposit savings to invalidate the high debt-to-income ratio and low balance checking. □

Extensions. The techniques in NEX can be applied to the following.

(1) Multi-class settings. NEX can explain multi-class $\mathcal{M}$. To this end, we replace $\neg\mathcal{M}(x_0)$ in RxGNNs with an ML predicate $\mathcal{M}(x_0) = i$, where $i \in [1, K]$ denotes the predicted class of x_0 (one of K distinct classes) by $\mathcal{M}$; binary classification is a special case with $K = 2$. In this setting, counterfactual explanations change the predicted class of x_0 from i to another class. We will test this setting in Section 6.

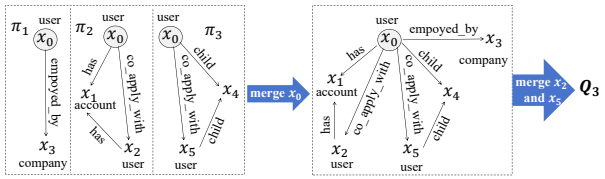


Figure 4: An example of how Q_3 is composed

(2) Positive outcomes. NEX can be extended to explain positive outcomes, but users typically care more about reversing unfavorable ones (*e.g.,* loan denials). For favorable results, factual explanations are usually preferred [23]. Thus, we focus on negative outcomes.

4 MODEL-GUIDED RULE DISCOVERY

To faithfully explain the bad outcome of a GNN, we propose model-guided rule discovery, based on a notion of motifs. We formulate the problem (Section 4.1) and show how to identify and compose motifs (Section 4.2), followed by the discovery algorithm (Section 4.3).

4.1 The Discovery Problem

Given an RxGNN $\varphi = Q[\bar{x}, x_0](X \to \neg\mathcal{M}(x_0))$ and a graph G, we consider the following criteria [23, 29] to evaluate the quality of φ.

Support. This is to measure how often an RxGNN φ can be applied in graph G. More specifically, the *support* of φ in G is defined as:

$$\text{supp}(\varphi, G) = \|Q(\bar{x}, x_0, G, X \wedge \neg\mathcal{M}(x_0))\|.$$

Here $Q(\bar{x}, x_0, G, X \wedge \neg\mathcal{M}(x_0))$ denotes the set of vertices $h(x_0)$ for all matches h of Q in G such that $h \models X \wedge \neg\mathcal{M}(x_0)$. Intuitively, the higher $\text{supp}(\varphi, G)$ is, the more frequent φ can be applied to G.

In particular, when $\varphi = Q[\bar{x}, x_0](\emptyset \to \neg\mathcal{M}(x_0))$ (*i.e.,* X is empty), we also denote $\text{supp}(\varphi, G)$ by $\text{supp}(Q, G)$ for simplicity.

For an integer σ, an RxGNN φ is σ-frequent if $\text{supp}(\varphi, G) \geq \sigma$.

Anti-monotonicity. Given RxGNNs $\varphi = Q[\bar{x}, x_0](X \to \neg\mathcal{M}(x_0))$ and $\varphi = Q'[\bar{x}, x_0](X' \to \neg\mathcal{M}(x_0))$, we say that pattern Q *subsumes* Q', denoted as $Q \ll Q'$, if for any graph G, if Q' has matches in G, then Q also has matches in G, *e.g.,* Q removes edges or vertices in Q'.

Similarly, we say that φ *subsumes* φ', denoted as $\varphi \preceq \varphi'$, if either $Q \ll Q'$ or $X \subseteq X'$, *i.e.,* φ has a less restrictive pattern or precondition than φ'. This results in the well-known *anti-monotonicity* property, *i.e.,* if $\varphi \preceq \varphi'$, then for any graph G, $\text{supp}(\varphi, G) \geq \text{supp}(\varphi', G)$.

Confidence. It measures how strong the connection between precondition X and negative prediction $\neg\mathcal{M}(x_0)$ is, formally defined as:

$$\text{conf}(\varphi, G) = \frac{\text{supp}(\varphi, G)}{\|Q(\bar{x}, x_0, G, X)\|}.$$

For a threshold δ, an RxGNN φ is δ-confident if $\text{conf}(\varphi, G) \geq \delta$. Unlike support, confidence does not have the anti-monotonicity.

RxGNNs of interest. Given a GNN-based model $\mathcal{M}$ and a graph G, denote by Σ_A the set of all σ-frequent and δ-confident RxGNNs on G pertaining to $\mathcal{M}$. A *cover* Σ of Σ_A is a minimal subset satisfying: (a) every rule φ in Σ_A is logically implied by Σ (written $\Sigma \models \varphi$, *i.e.,* for all graph G', if $G' \models \Sigma$ then $G' \models \varphi$); and (b) no rule φ in Σ is redundant (*i.e.,* $\Sigma \setminus \{\varphi\} \not\models \varphi$). That is, Σ includes only nontrivial and non-redundant RxGNNs. Moreover, we want RxGNNs composed from a set $\mathcal{S}_\mathcal{M}$ of *motifs* that play essential roles in $\mathcal{M}$'s predictions.

Motif. A (negative) motif is a graph pattern $\pi[\bar{x}, x_0]$ with a designated vertex x_0 that appears more frequently in the negative class than in the positive class in G. Formally, given a threshold η, a pattern $\pi[\bar{x}, x_0]$ is a *motif* if $\frac{n_t}{n_f} < \eta$, where n_t (resp. n_f) is the number

of vertices $h(x_0)$ for matches h of $\pi[\bar{x}, x_0]$ in G with $\mathcal{M}(h(x_0))$ = true (resp. false). Given an integer k, a motif is *k-bounded* if $|\bar{x}| \leq k$.

Motif composition. We can *compose* multiple motifs into graph patterns $Q[\bar{x}, x_0]$, via a *composition function* $\otimes(\cdot)$. Given two motifs $\pi_1[\bar{x}_1, x_0]$ = (V_1, E_1, L_1) and $\pi_2[\bar{x}_2, x_0]$ = (V_2, E_2, L_2), a pattern $Q[\bar{x}, x_0]$ = (V_Q, E_Q, L_Q) is composed of π_1 and π_2 via $\otimes(\cdot)$, if V_Q (resp. E_Q) is a *revision* of $V_1 \cup V_2$ (resp. $E_1 \cup E_2$), such that (a) the designated vertices x_0 from π_1 and π_2 are merged into a single x_0 in Q, and (b) if a vertex v_1 in V_1 and a vertex v_2 in V_2 bear the same label, they are either represented as distinct vertices or a merged vertex in V_Q, depending on how $\otimes(\cdot)$ specifies; and L_Q inherits the label assignment from π_1 and π_2. This extends naturally to the composition of a set S of motifs; we denote the set of patterns composed from S by $\otimes(S)$, with one or more patterns. As an example, Q_3 in Example 4 is composed from three motifs π_1, π_2, π_3 in Figure 4.

Model-guided discovery. We formalize the problem as follows.

- *Input*: A graph G, a GNN model $\mathcal{M}$, a set $\mathcal{S}_\mathcal{M}$ of k-bounded motifs, a composition function $\otimes(\cdot)$, a maximum number N of pattern vertices, a support threshold σ and a confidence threshold δ.
- *Output*: A cover Σ of RxGNNs Σ_A such that for each $\varphi \in \Sigma_A$, (a) $\text{supp}(\varphi, G) \geq \sigma$, (b) $\text{conf}(\varphi, G) \geq \delta$, (c) $Q[\bar{x}, x_0]$ of φ is composed from motifs in $\mathcal{S}_\mathcal{M}$ via $\otimes$, *i.e.,* $Q \in \cup_{S \subseteq \mathcal{S}_\mathcal{M}} \otimes(S)$, and (d) $|\bar{x}| \leq N$.

Here N is mostly to control the cost of discovery. Although one can plug in any $\mathcal{S}_\mathcal{M}$ and $\otimes(\cdot)$, we present our strategies for identifying and composing motifs in Section 4.2, targeted for $\mathcal{M}$'s explanation.

4.2 Motif Identification and Composition

Identifying a good set of motifs and defining an effective composition function $\otimes(\cdot)$ are challenging tasks for the following reasons.

(1) *Exponential motifs.* We want to identify discriminative structures that are decisive for $\mathcal{M}$'s predictions. This, however, can be exponential in the worst case. We need to strike a balance between the cost of generation and the discriminability of the motifs identified.

(2) *Excessive compositions.* Motif composition is essentially a combinatorial optimization problem with exponential cost, *e.g.,* if we use a naive $\otimes_{\text{all}}(\cdot)$ and there are M pairs of vertices with the same label in S, $\otimes_{\text{all}}(S)$ can have $O(2^M)$ patterns, most of which are useless.

To tackle these, we develop bounded motif identification with pre-pruning and effective composition by reinforcement learning.

Motif identification. Following the "explain-and-summarize" paradigm as in [16, 23], our strategy works in the following two steps.

(1) Explain: In the "explain" step, we extract a *concise* explanation subgraph $G_{\text{sub}}(v)$ for each negative vertex v in G (*i.e.,* $\mathcal{M}(v)$ = false), such that $\mathcal{M}$ could *reproduce* its prediction in $G_{\text{sub}}(v)$. The subgraphs for all negative vertices in G are stored in a set $\mathcal{G}$.

(2) Summarize: We abstract the common structures as motifs from subgraphs in $\mathcal{G}$, such that each motif is σ-frequent and k-bounded.

To support this, we use two primitive operators, described below.

Extractor. The extractor is an operator that extracts a subgraph for v where $\mathcal{M}$ can reproduce its prediction. A naive extractor is to compute the L-hop neighborhood graph of v, since the embedding of v is refined by aggregating from up to its L-hop neighbors via internal message passing, regardless of how large G is. In practice, however, this can be supported by established solutions, *e.g.,* [60, 96, 99].

Pattern generator. We use a pattern generator as the second operator to generate a set of candidate (σ-frequent and k-bounded) patterns from the subgraphs in $\mathcal{G}$, to be further verified. This is also known as a pattern mining problem, where existing solutions can be applied.

Algorithm. Given as input a graph G, a GNN model $\mathcal{M}$, a support threshold σ, the maximum size k and a threshold η, we outline the algorithm in Figure 5. Initializing $\mathcal{S}_\mathcal{M} = \emptyset$ (line 1), we extract a subgraph for each negative vertex $v \in G$ via the primitive extractor (lines 2-3). From these subgraphs, we mine σ-frequent and k-bounded patterns π via the primitive generator (line 4). Each candidate π is then evaluated by comparing its occurrence across positive and negative classes (lines 5-8). More specifically, we calculate the ratio $\frac{n_t}{n_f}$ of π, where n_t and n_f are defined in Section 4.1. We add π to $\mathcal{S}_\mathcal{M}$ if $\frac{n_t}{n_f} < \eta$ and π is not subsumed by any existing π' in $\mathcal{S}_\mathcal{M}$ (lines 7-8); intuitively, this is to avoid redundant motifs in $\mathcal{S}_\mathcal{M}$.

Implementation of primitive operators. A good extractor should extract concise subgraphs that preserve critical message flows that lead the model to its negative predictions. To obtain concise subgraphs, we apply a graph mask over edges following [96]. Then the pattern generator mines motifs on masked subgraphs, instead of the original L-hop neighborhood graphs, so as to efficiently identify most discriminative structures, since unimportant edges are already masked. We adopt gSpan [95] to mine motifs for its widespread use.

Remark. We only focus on mining σ-frequent and k-bounded motifs.

(1) The use of σ is to prune useless motifs without missing any rule above the support threshold. Specifically, given a motif π and a pattern Q composed from π (possibly with other motifs), π subsumes Q and thus, $\text{supp}(\pi, G) \geq \text{supp}(Q, G)$ by anti-monotonicity. Therefore, if a motif is not σ-frequent, all patterns composed from it will not be σ-frequent, which can be safely excluded from consideration.

(2) We use k to balance cost and discriminability: if k is too small, the motifs may not be discriminative enough; and if k is too large, it becomes costly to obtain all k-bounded motifs. In practice, it often suffices to set k small (*e.g.,* $k \leq 5$), *e.g.,* the best performance of a GNN-based model is observed when it aggregates information from its 2-hop neighbors [93], and "deeper versions of the model that, in principle, have access to more information, perform worse [55]".

Example 6: In Figure 4, we identify three 3-bounded motifs, π_1, π_2 and π_3, which characterize user x_0 from the perspectives of employment ties, financial situation and social connections, receptively. □

Composition function. As discussed in [54], to avoid the excessive compositions, we can learn a robust policy that heuristically guides the composition, to keep only the most promising pattern in $\otimes(S)$ (if it exists). The input/output of our $\otimes(\cdot)$ is as follows.

- *Input*: A set S of motifs from $\mathcal{S}_\mathcal{M}$ and a support threshold σ.
- *Output*: A pattern $Q^*[\bar{x}, x_0]$ such that $Q^* = \arg\max\{\text{conf}(Q) \mid Q \in \otimes_{\text{all}}(S)$ and Q is a σ-frequent pattern$\}$ (if it exists).

Our policy can be extended to keep multiple top-ranked patterns.

Motif composition can lead to a large state space. To this end, we adopt deep Q-learning (DQN) [64] to learn the policy since it is good at handling large state spaces and modeling complex structures.

More specifically, it takes the current pattern Q_i as a state s_i and each pair (v, v') of vertices in Q_i bearing the same label as an action

Input: G, $\mathcal{M}$, σ as above, a maximum size k and a threshold η.
Output: A set $\mathcal{S}_\mathcal{M}$ of motifs, where each motif is σ-frequent and k-bounded.
1. $\mathcal{S}_\mathcal{M} := \emptyset$; $\mathcal{G} := \emptyset$;
2. **for** each vertex $v \in G$ such that $\mathcal{M}(v) = \text{false}$ **do**
3. $\quad G_{\text{sub}}(v) := \text{extractor}(G, v)$; $\mathcal{G} := \mathcal{G} \cup \{G_{\text{sub}}(v)\}$;
4. $\mathcal{F} := \text{generator}(\mathcal{G}, \sigma, k)$;
5. **for** each candidate pattern $\pi[\bar{x}, x_0] \in \mathcal{F}$ **do**
6. $\quad n_t := \|Q(\bar{x}, x_0, G, \mathcal{M}(x_0))\|$; $n_f := \|Q(\bar{x}, x_0, G, \neg\mathcal{M}(x_0))\|$;
7. $\quad$ **if** $\frac{n_t}{n_f} < \eta$ and $\nexists \pi' \in \mathcal{S}_\mathcal{M}$ such that $\pi' \ll \pi$ **then**
8. $\quad\quad \mathcal{S}_\mathcal{M} := \mathcal{S}_\mathcal{M} \cup \{\pi[\bar{x}, x_0]\}$;
9. **return** $\mathcal{S}_\mathcal{M}$;

Figure 5: Algorithm getMotifs

a. We transit to the next state s_{i+1} by taking a and get a reward r:

$$r = \begin{cases} -1 & \text{if } \text{supp}(Q_{i+1}, G) < \sigma, \\ \text{conf}(Q_{i+1}, G) - \text{conf}(Q_i, G) & \text{if } \text{supp}(Q_{i+1}, G) \geq \sigma. \end{cases}$$

We also use a special action [end] to stop if the $\text{supp}(Q, G) < \sigma$.

To utilize DQN for composition, we adopt several component networks: (a) a *state encoder*, $\text{encoder}_\phi(Q)$ (resp. *action* $\text{encoder}_\phi(v, v')$), with parameter ϕ that maps a state $s = Q$ (resp. an action $a = (v, v')$)) to an embedding $\mathbf{e}_s$ (resp. $\mathbf{e}_a$) for learning, (b) a *Q-value network* $\mathbb{Q}_\theta(s, a)$ with parameter θ that predicts the expected accumulated reward (*i.e.,* Q-value) by taking a at s, (c) a *target network* $\mathbb{Q}_{\theta'}$, periodically copied from $\mathbb{Q}_\theta$ to provide stable training target. For limited space, network details are presented in [4]. The learning method and loss function are the same as classic DQN (see [4])

Given the trained $\mathbb{Q}_\theta(s, a)$, our composition function $\otimes(S)$ works as follows. We start with an initial pattern Q_0 (created by merging x_0 of all motifs in S into one x_0), and iteratively merge vertices with the maximum Q-value, until the termination condition is satisfied.

Example 7: Figure 4 illustrates how pattern Q_3 in Figure 2(b) is composed. Initially, the designated x_0's of all motifs are merged into a single vertex. Then the policy merges x_2 and x_5 (both are user vertices), followed by the special termination action [end]. □

4.3 Parallel Discovery Algorithm

We now present RxGNNMiner, our parallel discovery algorithm. Its novelty includes (a) the use of motifs and composition functions to obtain desired graph patterns, (b) a novel predicate ordering scheme for precondition generation, facilitating multi-thread parallelism.

Algorithm. As shown in Figure 6, RxGNNMiner first initializes the set C of graph patterns and the result set Σ to be empty (line 1). Then it composes the motifs in $\mathcal{S}_\mathcal{M}$ into patterns in C using the composition function $\otimes(\cdot)$ (line 2), via procedure genPatterns (lines 15-18). Given as input a set S_{sel} of selected motifs, a set S_{re} of remaining motifs, function $\otimes(\cdot)$ and threshold σ, genPatterns conducts depth-first search. In each round, we check whether $Q = \otimes(S_{\text{sel}})$ is σ-frequent (line 15). If so, we add Q into C (line 16) and recursively call genPatterns by adding a motif π from S_{re} to S_{sel} to find more frequent patterns (lines 17-18). If Q is not σ-frequent, the recursion stops and the current C is directly returned since exploring more motifs can only lower the support by anti-monotonicity. In the first call of genPatterns, we set $S_{\text{sel}} = \emptyset$ and $S_{\text{re}} = \mathcal{S}_\mathcal{M}$ (line 2).

For each resulting Q in C, we compute X in a levelwise manner (lines 3-13). More specifically, we first generate the set $\mathcal{P}$ of potential predicates guided by $\mathcal{M}$ via procedure genPredicates (line 4, see below). Then we start with $X = \emptyset$ (line 5) and iteratively expand X with more predicates in $\mathcal{P}$ (lines 6-13), by maintaining a work queue

Algorithm RxGNNMiner
Input: G, $\mathcal{M}$, $\mathcal{S}_\mathcal{M}$, $\otimes(\cdot)$, N, σ, δ as stated above.
Output: A set Σ of RxGNNs pertaining to model $\mathcal{M}$.
1. $C := \emptyset$; $\Sigma := \emptyset$;
2. $C := \text{genPatterns}(\emptyset, \mathcal{S}_\mathcal{M}, C, \otimes, \sigma)$; /* Pattern generation*/
3. **for** each pattern Q in C **do** /* Precondition generation*/
4. $\quad \mathcal{P} := \text{genPredicates}(Q, \mathcal{M})$;
5. $\quad \Phi :=$ an empty work queue; $X := \emptyset$, $\Phi.\text{add}(\langle X, \mathcal{P}\rangle)$;
6. $\quad$ **while** $\Phi \neq \emptyset$ **do**
7. $\quad\quad \langle X, \mathcal{P}\rangle := \Phi.\text{pop}()$; $\varphi := Q[\bar{x}, x_0](X \to \neg\mathcal{M}(x_0))$;
8. $\quad\quad$ **if** φ is σ-frequent and δ-confident **then** /*Checked at each thread*/
9. $\quad\quad\quad \Sigma := \Sigma \cup \{\varphi\}$;
10. $\quad\quad$ **elseif** φ is σ-frequent but not δ-confident **then**
11. $\quad\quad\quad$ **for** each p in $\mathcal{P}$, ordered by perplexity-based ranking scores **do**
12. $\quad\quad\quad\quad \mathcal{P}_{\text{new}} :=$ the set of all predicates in $\mathcal{P}$ ordered after p;
13. $\quad\quad\quad\quad \Phi.\text{add}(\langle X \cup \{p\}, \mathcal{P}_{\text{new}}\rangle)$;
14. **return** $\text{cover}(\Sigma)$;

Procedure genPatterns
Input: A set S_{sel} of selected motifs, a set S_{re} of remaining motifs the can be selected, the current set C of graph patterns, $\otimes(\cdot)$, and σ as above.
Output: An updated set C of graph patterns.
15. **if** $Q = \otimes(S_{\text{sel}})$ is σ-frequent **then**
16. $\quad C := C \cup \{Q\}$;
17. $\quad$ **for** each π in S_{re} **do**
18. $\quad\quad C := C \cup \text{genPatterns}(S_{\text{sel}} \cup \{\pi\}, S_{\text{re}} \setminus \{\pi\}, C, \otimes, \sigma)$;
19. **return** C;

Figure 6: Algorithm RxGNNMiner

Φ (line 5). Each pair $\langle X, \mathcal{P}\rangle$ in Φ forms a computational task. If $\varphi = Q[\bar{x}, x_0](X \to \neg\mathcal{M}(x_0))$ is σ-frequent and δ-confident, it is valid to be included in Σ and there is no need to expand X since we only want "minimal" RxGNNs (lines 8-9). Otherwise, if φ is σ-frequent but not δ-confident, we expand X by adding a new predicate from $\mathcal{P}$ into X (lines 10-13), leading to a new task to be added to Φ. Unlike conventional levelwise expansion [28], we order predicates in $\mathcal{P}$ based on perplexity-based scores (line 11, see below). When φ is not σ-frequent, we can directly skip its subsequent expansion, since all RxGNNs expanded from φ are not σ-frequent by anti-monotonicity. Finally, when all tasks in Φ are done, a set of Σ of non-redundant is returned via *cover computation* [29] (line 14, see below).

Example 8: Assume that for Q_3, $\mathcal{P}$ has all predicates in X_3 (Example 4), ordered by the occurrence. Initially, $X = \emptyset$ and we check $\varphi' = Q_3(\emptyset \to \neg\mathcal{M}_3(x_0))$. If φ' is σ-frequent but not δ-confident, we expand X with the next predicate $x_0.\text{debt\_to\_income} \geq 40\%$ and check the rule again. This process continues until φ_3 is found. □

Procedure genPredicates. One can follow [28] to generate all possible predicates as $\mathcal{P}$ for a given Q. This, however, is over-demanding. In light of this, we only keep predicates p closely related to $\neg\mathcal{M}$, *i.e.,* p is kept in $\mathcal{P}$ if it is satisfied more frequently in the matches h of Q from negative classes than positive classes, *w.r.t.* a threshold (*e.g.,* η).

We separate pattern mining and predicate generation to ensure that no needed RxGNNs are missed due to the premature binding of patterns and predicates, such as potential predicates across motifs.

Predicate ordering. We evenly distribute workloads to processors for parallel processing [25]. Following [23], we use multiple threads for precondition generation, where each thread *independently* handles a task $\langle X, \mathcal{P}\rangle$. Here the processing order of predicates is important.

To illustrate, consider two tasks $t_1 = \langle X_1, \mathcal{P}_1\rangle$ and $t_2 = \langle X_2, \mathcal{P}_2\rangle$ for Q where $X_1 = \{p_1\}$ and $X_2 = \{p_2\}$. Denote the RxGNNs for t_1

and t_2 by ϕ_1 and ϕ_2, respectively, and assume that ϕ_1 is σ-frequent but ϕ_2 is not. If t_1 is processed before t_2, we may expand $X_1 = \{p_1\}$ to new $X_{12} = \{p_1, p_2\}$, yielding a new task $\langle X_{12}, \mathcal{P}_{12}\rangle$. However, we do not need to handle this task if we process t_2 first, since ϕ_2 is not σ-frequent. This motivates us to develop effective predicate ordering so that discriminative and cheap predicates are expanded early.

To measure the discriminability of predicates for negative predictions of $\mathcal{M}$ at x_0 (*i.e.,* $\neg\mathcal{M}(x_0)$), we leverage large language models (LLMs) for its rich prior knowledge and its unique capability in contextual understanding. For a predicate p, we define its perplexity:

$$\mathsf{PPL}(p) \propto \exp\left(-\log P_{\mathrm{LLM}}(\mathsf{nlp}(p) \mid \mathsf{nlp}(Q))\right),$$

where $\mathsf{nlp}(\cdot)$ is natural language description of Q or p, and P_{LLM} is a probability given by LLMs, indicating the likelihood that appending $\mathsf{nlp}(p)$ into $\mathsf{nlp}(Q)$ yields a statement consistent with $\neg\mathcal{M}(x_0)$.

Example 9: Consider two predicates p_1:x_0.debt_to_income $\geq$ 40% and p_2:x_0.gender = Male. The simplified natural language description of Q_3 appended with p_1 (resp. p_2) is T_1 ="Given the topology in Q_3, x_0's loan is denied since his debt to income ratio exceeds 40%" (resp. T_2 ="…, x_0's loan is denied since he is a male"). Since T_1 is more reasonable, the LLM will assign lower perplexity to p_1. □

Intuitively, $\mathsf{PPL}(p)$ quantifies how correlated a predicate p is to $\neg\mathcal{M}(x_0)$; a lower $\mathsf{PPL}(p)$ indicates that the LLM is more confident about such correlation and thus, it should be processed with higher priority. Besides, we also consider the time cost of validation. Thus, the ranking score of p is $\mathsf{PPL}(p) \times \mathsf{time}(p)$, where $\mathsf{time}(p)$ is the evaluation time of p estimated on sample data, and we process predicates in the ascending order of ranking scores. *i.e.,* we favor predicates strongly correlative to $\neg\mathcal{M}$, while penalizing expensive ones.

Cover computation. We compute a cover Σ of σ-frequent and δ-confident RxGNNs by adapting the implication analysis [28] and parallel techniques [29] to RxGNNs (see details in [4]).

Remark. To select appropriate parameters (*e.g.,* thresholds σ and δ, and motif size k) for discovery, users can leverage data statistics to set initial parameters to enable quick discovery of an initial rule set, and iteratively fine-tune the parameters based on insights from this batch using incremental rule discovery [15]. Incremental techniques in [15] naturally extend to the discovery of RxGNNs.

Complexity. Algorithm RxGNNMiner takes at most $O(\frac{1}{n} \times |\mathcal{S}_{\mathcal{M}}|^K \sum_{X\in \mathsf{pow}(\mathcal{P})} |G|^{|X|})$ time with n processors, where K is the maximum number of motifs in $\mathcal{S}_{\mathcal{M}}$ that can compose a σ-frequent pattern Q ($K \ll |\mathcal{S}_{\mathcal{M}}|$) and $\mathcal{P}$ is the set of all potential predicates defined on a pattern Q and $\mathsf{pow}(\mathcal{P})$ is the power set of $\mathcal{P}$. This is because (a) the computation is dominated by the validation of support and confidence, where the computational tasks are evenly distributed across n processors, and (b) it examines $\mathsf{pow}(\mathcal{P})$ to generate X's for each Q at worst. This is the inherent cost for rule discovery [18, 25, 28, 49]. This said, we show that RxGNNMiner is parallelly scalable [56], reducing runtime nearly n-fold with n processors (see [4] for details; the space cost is also given in [4]).

5 LOCAL EXPLANATIONS

This section develops an algorithm to identify counterfactual explanations for negative GNN predictions. The problem is as follow.

- *Input*: A graph G, a GNN-based classifier $\mathcal{M}$, the set Σ of RxGNNs, and a negative vertex u in G with $\mathcal{M}(u) = \mathsf{false}$.
- *Output*: The minimum set $\mathbb{O}$ of graph perturbations such that no $\varphi \in \Sigma$ is applicable at u in $G \ominus \mathbb{O}$. That is, after updating G with $\mathbb{O}$, the prediction $\mathcal{M}(u)$ is swapped from false to true.

A brute-force way is to compute the set $\mathbb{H}_u(\varphi, G)$ of all witnesses h of φ pivoted at u (*i.e.,* $h \models \varphi$ and $h(x_0) = u$) for each φ in Σ. Then it identifies the set $\mathbb{O}$ so that for *each* h in $\mathbb{H}_u(\varphi, G)$, h no longer satisfies X or Q of φ in $G \ominus \mathbb{O}$. This is too costly since it has to enumerate all witnesses h of each φ in Σ, while even checking the existence of matches of a general pattern Q is already NP-hard [38] (see below).

Note that a key step to compute the matches of $Q[\bar{x}, x_0]$ is vertex filtering, which selects candidate vertices in G for each pattern vertex $x \in \bar{x}$ and eliminates those that cannot be mapped to x in *any* match. Formally, given a pattern vertex x, we define $C_Q(x)$ to be the candidate set of potential vertices in G that can be mapped to x in a match of Q. We require $C_Q(x)$ to have the following properties.

- *General-pattern support.* It is defined on general patterns Q, not just star patterns [23] or patterns with bounded tree-width [7].
- *PTIME computation.* Although computing exact matches of general Q in NP-hard, we require $C_Q(x)$ to be computed in PTIME.
- *Soundness.* Set $C_Q(x)$ is *sound* if for each match h of Q where $h(x) = v, v \in C_Q(x)$. As a compromise, $C_Q(x)$ may have *false positives*, *i.e.,* for $v \in C_Q(x)$, there may exist no h of Q such that $h(x) = v$.

For example, a naive $C_Q(x)$ can be V (*i.e.,* all vertices in G). Nevertheless, a good $C_Q(x)$ should be as *tight* as possible, with few false positives, in order to offer a concise encapsulation of the search space.

Below we extend vertex filtering to compute *candidate sets* for RxGNNs, which in addition to Q, also satisfy X (Section 5.1). Based on this, we develop a strategy to find perturbations (Section 5.2).

5.1 Candidate sets for RxGNNs

We start with the definition of candidate sets for RxGNNs.

Candidate sets. Along the same line as $C_Q(x)$ for pattern Q, given an RxGNN $\varphi = Q[\bar{x}, x_0](X \to \neg\mathcal{M}(x_0))$ and a vertex u in G, we define $C^u_\varphi(x)$ to be the candidate set of potential vertices in G that can be mapped to x in a witness h of φ pivoted at u (*i.e.,* h is a match of pattern Q, $h \models \varphi$ and $h(x_0) = u$). When u is clear in the context, we simply write $C^u_\varphi(x)$ as $C_\varphi(x)$. Denote the union set of $C_\varphi(x)$ for all $x \in \bar{x}$ as *the candidate space* C_φ. We define *sound* C_φ as above.

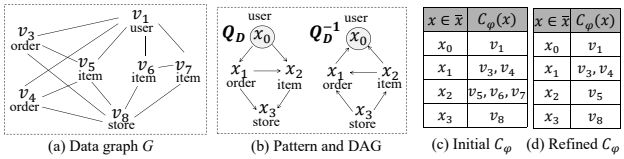
Sufficient condition. Inspired by *emptyset failure* [43], we develop a sufficient condition on sound C_φ for checking the existence of witness for φ pivoted at $u \in G$ (*i.e.,* checking if φ is applicable at u).

Lemma 1: *Given a graph G, a vertex u in G, an RxGNN φ, and a sound candidate space C_φ of φ, if there exists a pattern vertex x of φ such that $C_\varphi(x) = \emptyset$, then there is no witness h of φ pivoted at u in G.* □

Proof. This is granted by the soundness of C_φ: assume that there is a witness h of φ pivoted at u in G, then by the soundness, $h(x) \in C_\varphi(x)$ for each $x \in \bar{x}$, contradicting to $C_\varphi(x) = \emptyset$ for some x. □

We say that a sound C_φ is *invalid* if it satisfies the condition in Lemma 1. Intuitively, for a vertex u in G, if C_φ is invalid, φ is not applicable at u. This suggests that for each φ in Σ, we can compute a sound C_φ for φ; if C_φ is not yet invalid, we update G (and hence C_φ) by graph perturbations on G until C_φ becomes invalid.

Example 10: Consider a graph G and an RxGNN φ with pattern Q_D in Figures 7(a)-(b) (only vertex labels are shown). Assume that φ has



(a) Data graph G (b) Pattern and DAG

$x \in \bar{x}$	$C_\varphi(x)$
x_0	v_1
x_1	v_3, v_4
x_2	v_5, v_6, v_7
x_3	v_8

(c) Initial C_φ

$x \in \bar{x}$	$C_\varphi(x)$
x_0	v_1
x_1	v_3, v_4
x_2	v_5
x_3	v_8

(d) Refined C_φ

Figure 7: Candidate space-based counterfactual explanation

two witnesses pivoted at v_1, *i.e.,* h_1: $(x_0, x_1, x_2, x_3) \mapsto (v_1, v_4, v_5, v_8)$ and h_2: $(x_0, x_1, x_2, x_3) \mapsto (v_1, v_3, v_5, v_8)$. A sound and valid C_φ is given in Figure 7(d), as there exists no x in $\bar{x}$ such that $C_\varphi(x) = \emptyset$. □

Problem. Based on these intuitions, it suffices to study a candidate space-based counterfactual explanation problem, stated as follows.

- *Input*: G, $\mathcal{M}$, Σ, and u with $\mathcal{M}(u) =$ false as above.
- *Output*: The minimum set $\mathbb{O}$ of graph perturbations on G such that for each $\varphi \in \Sigma$, C_φ is *invalid* after updating G with $\mathbb{O}$.

Below we present a PTIME method for computing C_φ.

Candidate space construction. To reduce perturbations for invalidating C_φ, we construct a tight C_φ by exploiting *DAG-graph dynamic programming* [43], to check both topological structures (*i.e.,* to match Q) and vertex attributes (*i.e.,* to satisfy X) simultaneously.

Overview. Given a graph G, a vertex u in G, an RxGNN $\varphi \in \Sigma$, we construct a candidate space C_φ pivoted at u in three steps.

(1) Rooted DAG. We first build a *rooted directed acyclic graph (DAG)* Q_D from the pattern $Q[\bar{x}, x_0]$ of φ, which (a) contains no cycle, and (b) in G_D, only the designated vertex x_0 has no incoming edges.

(2) Initialization. We initialize C_φ pivoted at u by checking labels, degrees and predicates, *i.e.,* $v \in C_\varphi(x)$ if $L_G(v) = L_Q(x)$, $\deg_G(v) \geq \deg_Q(x)$ and v satisfies all constant/1-WL predicates defined on x in X of φ, where deg is the degree in G/Q; in particular, $C_\varphi(x_0) = \{u\}$.

(3) Refinement. We refine C_φ iteratively via *DAG-graph dynamic programming*, which alternates *forward propagation* along Q_D and *backward propagation* along its reverse Q_D^{-1} until C_φ stabilizes.

Example 11: In Example 10, the pattern Q_D of φ is itself a DAG; it reverse Q_D^{-1} is also shown in Figure 7(b). The initial C_φ is given in Figure 7(c). After refinement, we obtain final C_φ in Figure 7(d). □

Below we present the details of DAG building and refinement.

DAG building. To build a DAG Q_D from Q, we treat the designated vertex x_0 as the root. We traverse Q in a BFS order from x_0 and direct all pattern edges from upper levels to lower levels. To process more selective vertices earlier, the directions between vertices at the same level are decided by an *infrequent-first* order: we group vertices by their labels (one group per label) and then sort the groups in the ascending order of their numbers of occurrence in G; within each group, we sort vertices in the descending order of degrees.

Refinement. This step refines C_φ by propagating topological and logical constraints in both forward and backward directions. It implements dynamic programming between DAG Q_D and graph G [43].

More specifically, to refine $C_\varphi(x)$, it computes an auxiliary set $C'_\varphi(x)$ by dynamic programming such that $v \in C'_\varphi(x)$ iff (a) $v \in C_\varphi(x)$, (b) there exists v_c adjacent to v such that $v_c \in C'_\varphi(x_c)$ for every child x_c of $x \in Q_D$, and (c) there exists v_p such that $v.A \oplus v_p.B$ and $v_p \in C'_\varphi(x_p)$ for every variable predicate $x.A \oplus x_p.B$ in X.

Based on this, we compute $C_\varphi(x)$ by alternating forward refinement (propagating along Q_D) and backward refinement (along the reverse Q_D^{-1} of Q_D). This bidirectional refinement continues until no changes can be made to the candidate sets. In practice, we can empirically set a maximum number of refinements, *e.g.,* when the filtering rate after the first three refinements is below 1% [43].

Input: A graph G, a model $\mathcal{M}$, a set Σ of RxGNNs and a vertex u.
Output: A set $\mathbb{O}$ such that for each φ in Σ, C_φ of φ is *invalid* on $G \ominus \mathbb{O}$.

```
1. 𝕆 := ∅;
2. for each RxGNN φ = Q[x̄, x_0](X → ¬M(x_0)) in Σ do
3.     C_φ := constructCand(u, φ, G);
4.     while C_φ is valid do
5.         v* := pickVertex(C_φ); // pick the vertex to be removed
6.         C_φ := refine(C_φ, v*); // compute the refined C_φ after removing v*
7.         𝕆 = 𝕆 ∪ {the perturbation that removes v* from C_φ};
8. return 𝕆;
```

Figure 8: Counterfactual algorithm CFE

Analysis. The candidate space C_φ has the following property.

Proposition 2: *Given a vertex u and an RxGNN φ, DAG-graph dynamic programming gives a sound candidate space C_φ in $O(|Q||G|)$ time,* i.e., *for each witness h of φ with $h(x) = v$, v must be in $C_\varphi(x)$.* □

Proof. We verify this along the same line as [43]. (1) DAG-graph dynamic program takes $O(|Q||G|)$ time. Note that the overall computation is dominated by pattern matching; the checking of predicates does not increase the complexity bound. (2) The soundness of C_φ is warranted by the following: (a) the initial candidate space is sound; (b) by induction, the auxiliary set $C'_\varphi(x)$ is computed correctly; and (c) if a vertex v does not satisfy the soundness condition, the candidate space C_φ is still sound after we remove v from $C_\varphi(x)$. □

5.2 Algorithm for Counterfactual Explanations

We develop an algorithm to compute counterfactual explanations.

Overview. Taking G, $\mathcal{M}$, a set Σ of RxGNNs and a vertex u in G as input, our algorithm, CFE, processes the RxGNNs in Σ one by one.

For each φ in Σ, we first construct its candidate space C_φ. Then, we iteratively identify a set of graph perturbations to make C_φ invalid (*i.e.,* $C_\varphi(x) = \emptyset$ for some x in $\bar{x}$). In each iteration, we select the vertex with the best *cost-effectiveness* to remove from $C_\varphi(x)$, which will trigger a refinement cascade to invalidate C_φ for φ.

This process continues until C_φ becomes invalid for *all* φ in Σ, and the union of all applied perturbations forms the desired set $\mathbb{O}$.

Below we present (a) a perturbation strategy, (b) a cost-effective way to decide the right vertex to remove, and (c) algorithm CFE.

Perturbation strategy. Given $v \in C_\varphi(x)$, there are two types of perturbations than can make v no longer map to x in *any* witness.

Feature modification. Note that a GNN typically aggregates more information from hub vertices. If v is a *hub vertex* in G (*i.e.,* its degree is above a threshold τ), we modify some important attributes A of v to make at least one predicate involving $x.A$ in X no longer satisfied.

To measure attribute importance, we adopt GINI values [22, 23]; it shows how well X divide the pivots into different predicted classes of $\mathcal{M}$ [12, 42]. Denote the GINI value with (resp. without) $x.A$ in X by $\mathsf{Gini}(X)$ (resp. $\mathsf{Gini}(X_{\mathsf{noA}})$). We define the importance of $x.A$ in X as $\Delta\mathsf{Gini}_{x.A} = \mathsf{Gini}(X_{\mathsf{noA}}) - \mathsf{Gini}(X)$, *i.e.,* the reduction of the GINI value by $x.A$. A larger value means that $x.A$ is more important.

Remark. Following [76], we want to ensure the *plausibility* (*i.e.,* make sense in the real world) and *feasibility* (*e.g.,* the *gender* of a user cannot be changed) of feature modification, We reuse the perplexity-based computation in Section 4.3 for this purpose (presented in [4]).

Edge removal. We remove incident edges of v such that all witnesses h with $h(x) = v$ no longer match Q of φ. To illustrate, assume *w.l.o.g.* that (x_p, l, x) is a selected edge in Q. We remove all edges (v_p, l, v) in G where $v_p \in C_\varphi(x_p)$. This strategy is suitable for lower-degree v.

Cost-effectiveness. To decide the right variable x and vertex $v \in C_\varphi(x)$ to perturb, we consider both the *cost* and the *effectiveness*.

Cost. According the two perturbation strategies above, the cost of applying a perturbation to vertex v in $C_\varphi(x)$ is defined as:

$$\text{cost}(v) = \begin{cases} \max\limits_{x.A \text{ in } X} \Delta\text{Gini}_{x.A}, & \text{if } X \neq \emptyset \text{ and } \deg_G(v) \geq \tau, \\ \beta|\{(v_p, l, v) \mid v_p \in C_\varphi(x_p)\}|, & \text{otherwise,} \end{cases}$$

where τ is the hub degree threshold and β is a balancing parameter.

Effectiveness. The removal of a vertex v from $C_\varphi(x)$ may trigger a cascade of vertex removals in other $C_\varphi(x')$ $(x \neq x')$ during refinement. We exploit this cascading effect, by defining the effectiveness of removing v on the *entire* candidate space C_φ, not just on the set $C_\varphi(x)$ where v is removed. More specifically, we compare the size of $C_\varphi(x)$ for *all* x in $\bar{x}$ (not just $C_\varphi(x)$ where v is removed), before and after the refinement triggered by the removal of v in $C_\varphi(x)$.

Formally, we define the *effectiveness* of removing v as:

$$\text{eff}(v) = \alpha^K \big(1 - \min_{x \in \bar{x}} \frac{|C_\varphi^\Delta(x)|}{|C_\varphi(x)|}\big),$$

where v is a K-hop neighbor of the pivot u, α is the hop-decay factor, and C_φ^Δ is the refined C_φ after v is removed. Intuitively, vertices closer to u are more important and thus should be removed with higher priority; the smaller a set $C_\varphi^\Delta(x)$, the better the effectiveness.

Taken together, the *cost-effectiveness* of each $v \in C_\varphi$ is $\frac{\text{eff}(v)}{\text{cost}(v)}$.

Algorithm. We present algorithm CFE in Figure 8. It returns a set $\mathbb{O}$ such that for each φ in Σ, C_φ of φ is *invalid* on $G \ominus \mathbb{O}$.

Initializing the set $\mathbb{O}$ to be empty (line 1), we process each φ in Σ one by one (lines 2-7). For each φ, we compute its candidate space C_φ pivoted at u via procedure constructCand (line 3, Section 5.1). As long as C_φ is still valid (line 4), *i.e.,* there exists no variable x in $\bar{x}$ such that $C_\varphi(x) = \emptyset$, we pick a vertex v^* in C_φ to be removed from C_φ based on the cost-effectiveness via procedure pickVertex (line 5, see below). The vertex v^* is then removed from C_φ and C_φ is refined accordingly (line 6, Section 5.1). The perturbations that remove v^* from C_φ are added to $\mathbb{O}$ (line 7). Eventually, all C_φ's for all $\varphi \in \Sigma$ become invalid and the set $\mathbb{O}$ of perturbations is returned (line 8).

Example 12: Continuing Example 10, we compute candidate space C_φ in Figure 7(d). Since C_φ is valid, we pick a vertex in C_φ to remove. Consider v_4 and v_5. If we remove v_4, the resulting C_φ (*i.e.,* $C_\varphi(x_0) = \{v_1\}$, $C_\varphi(x_1) = \{v_3\}$, $C_\varphi(x_2) = \{v_5\}$, $C_\varphi(x_3) = \{v_0\}$) is still valid, leading to additional perturbations to invalidate C_φ. In contrast, it is more effective to remove v_5 since it directly invalidates C_φ (*i.e.,* $C_\varphi(x_2) = \emptyset$ after removing v_5) and we finish the processing of φ. □

Procedure pickVertex. To decide the most cost-effective vertex to be removed from C_φ, one can compute exact cost/effectiveness of all vertices in C_φ; this, however, requires $O(|C_\varphi|)$ calls of refinements.

To reduce the cost, we adopt clustering. We reuse encoder$_\phi()$ in Section 4.2 to embed the L-hop neighborhood graph for each v in C_φ and cluster the embeddings into groups. Intuitively, vertices from the same group have similar neighborhood and thus, they are likely to have similar effects when removed from C_φ. For each group g, we select a representative vertex v_g and compute its cost-effectiveness.

Let g^* be the group whose v_{g^*} has the highest cost-effectiveness. Then for each $v \in g^*$, we compute the exact eff(v) and cost(v), and select the vertex with the true highest cost-effectiveness as v^*.

Complexity. The time cost of CFE is $O(|\Sigma||C_\varphi||Q||G|(\#\text{group} + |g_{\max}|))$, where #group is the number of groups, and $g_{\max}$ is the largest group, since for each φ in Σ, (a) C_φ is computed in $O(|Q||G|)$ time, and (b) $O(|C_\varphi|)$ vertices will be removed from C_φ, where each removal takes $O(\#\text{group} + |g_{\max}|)$ times of refinement to decide.

6 EXPERIMENTAL STUDY

Using real-life graphs, we empirically tested the accuracy and efficiency of counterfactual and global explanations generated by NEX.

Experimental setting. We start with the experimental setting.

Datasets. We used four real-life graphs G in Table 3 (cf. [4]): (1) Loan [2], a loan graph used to predict whether a user should be approved for loan; (2) Claim [101], a medical claim network dataset designed for insurance fraud detection; (3) Trans [1], a transactional network for financial fraud detection; and (4) Amazon [3], a multi-class e-commerce graph, where products are labeled with various classes.

GNN models. Following [96, 98, 99], we selected four GNN models for vertex classification on heterogeneous graphs: (1) GCN [55], a standard message-passing graph convolutional network; (2) GAT [84], which uses attention mechanism to improve vertex representations; (3) GIN [92], which effectively captures graph structural information; and (4) HGT [44], which designs a vertex- and edge-type dependent attention mechanism to handle graph heterogeneity. Unless stated explicitly, we used GCN as our default GNN model.

Baselines. We compared two baselines for both local and global explanations: (1) GVEX [16], which is originally designed for graph classification with a two-tier explanation structure; (2) CFGraph [5], which adopts a notion of counterfactual graphs for local explanations and aggregates multiple such graphs for global explanations.

For local explanations, we tested three more baselines: (3) CF-Exp (*i.e.,* CF-GNNExplainer) [59], (4) CLEAR [63], and (5) InduCE [85], which give counterfactual explanations by optimization, graph variational autoencoder, and reinforcement learning, respectively.

For global explanations, we additionally tested: (6) XGNN [97], which optimizes inputs to maximize the prediction confidence of a target class, (7) GNNInt (*i.e.,* GNNInterpreter [88]), which generates global explanations by a numerical optimization approach. (8) LMiner, a variant of RxGNNMiner, which mines RxGNNs via traditional levelwise search; we only compared LMiner for efficiency.

We adapted all baselines to vertex classification on heterogeneous graphs where vertices carry various attributes, and optimized all counterfactual methods to minimize perturbations (see [4]).

Rules. For the four GNNs, we discovered on average 194, 357, 168 and 562 RxGNNs on Loan, Claim, Trans and Amazon, respectively.

Default parameters. We set the support threshold σ (resp. confidence δ) as 3K, 8K, 1K and 1K (resp. 0.7, 0.6, 0.7 and 0.5) on Loan, Claim, Trans and Amazon, respectively. We set (a) $k = 5$ (resp. $N = 15$) for the maximum size of motifs (resp. patterns), (b) $\eta = 0.6$ for motif identification, (c) $\alpha = 0.8$ for hop-decay, (d) $\beta = 0.01$ for the balancing parameter, and (e) $n = 8$ for the number of processors.

GNN	Method	Loan			Claim			Trans			Amazon		
		fidelity	sparsity	time(s)	fidelity	sparsity	time(s)	fidelity	sparsity	time(s)	fidelity	sparsity	time(s)
GCN	GVEX	0.836	5.6%	367.34	0.615	4.3%	492.65	0.754	3.9%	432.33	0.693	5.3%	2587.5
	CFGraph	0.454	17.1%	3.45	0.441	15.5%	4.22	0.481	10.8%	3.30	0.655	10.2%	2.13
	CF-Exp	0.486	23.1%	1.66	0.383	14.7%	2.75	0.56	8.1%	1.99	0.683	11.1%	1.82
	Clear	0.653	10.2%	10.77	0.517	13.4%	15.54	0.635	7.1%	10.11	0.679	9.7%	24.37
	InduCE	0.687	13.7%	5.31	0.573	8.9%	6.81	0.703	5.3%	4.52	0.684	8.4%	4.69
	NEX	**0.935**	**3.3%**	**0.97**	**0.719**	**2.6%**	**0.74**	**0.809**	**2.2%**	**1.36**	**0.734**	**3.3%**	**1.19**
GAT	GVEX	0.843	5.3%	355.53	0.618	4.5%	485.67	0.748	3.9%	444.27	0.695	4.9%	2629.1
	CFGraph	0.456	18.2%	3.67	0.437	15.9%	4.91	0.466	12.3%	3.16	0.632	10.2%	2.73
	CF-Exp	0.510	18.4%	2.34	0.406	14.5%	3.52	0.555	9.7%	2.58	0.688	10.4%	1.89
	Clear	0.739	10.8%	14.44	0.526	14.7%	20.41	0.674	6.3%	12.48	0.689	9.1%	28.84
	InduCE	0.706	9.7%	5.57	0.619	8.2%	8.74	0.671	5.5%	4.42	0.672	8.1%	4.40
	NEX	**0.936**	**2.9%**	**0.87**	**0.694**	**3.5%**	**0.84**	**0.806**	**3.1%**	**1.51**	**0.729**	**2.9%**	**1.34**
GIN	GVEX	0.855	5.6%	403.43	0.626	4.5%	440.83	0.726	4.1%	499.90	0.685	5.5%	2632.4
	CFGraph	0.462	17.7%	4.25	0.455	14.4%	5.15	0.459	10.8%	4.50	0.651	9.9%	2.17
	CF-Exp	0.508	20.4%	2.57	0.375	16.2%	2.79	0.538	9.7%	2.40	0.680	10.6%	1.93
	Clear	0.663	15.3%	12.93	0.509	10.9%	17.23	0.641	5.4%	11.98	0.676	10.2%	22.68
	InduCE	0.638	13.4%	6.14	0.595	9.6%	8.86	0.629	5.8%	4.95	0.677	8.8%	4.29
	NEX	**0.927**	**3.5%**	**1.00**	**0.690**	**3.1%**	**0.80**	**0.764**	**3.5%**	**1.29**	**0.715**	**3.8%**	**1.14**
HGT	GVEX	0.839	5.1%	588.31	0.611	4.8%	695.64	0.735	3.9%	601.78	0.688	5.3%	3938.9
	CFGraph	0.453	15.6%	5.34	0.449	16.2%	6.20	0.469	13.3%	5.55	0.643	11.5%	3.52
	CF-Exp	0.505	20.5%	2.88	0.405	16.6%	5.22	0.542	9.2%	3.56	0.692	12.3%	2.15
	Clear	0.684	14%	14.43	0.512	10.5%	19.14	0.659	6.8%	14.71	0.695	9.2%	34.98
	InduCE	0.671	12.2%	6.99	0.602	9.2%	10.94	0.685	5.9%	6.27	0.684	9.1%	5.61
	NEX	**0.929**	**3.5%**	**0.92**	**0.693**	**3.3%**	**0.96**	**0.797**	**3.3%**	**1.36**	**0.733**	**3.5%**	**1.38**

Table 2: Local (counterfactual) explanations

Dataset	$\|V\|$	$\|E\|$	#attr	#vertex labels	#classes
Loan	504K	1008K	13	5	2
Claim	702K	1674K	16	4	2
Trans	15.4M	12.7M	10	2	2
Amazon	1.7M	2.5M	15	3	5

Table 3: Dataset statistic (details in [4])

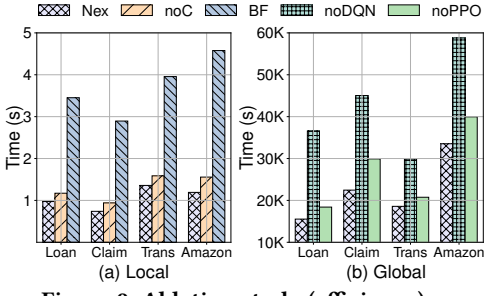


Figure 9: Ablation study (efficiency)

Environment. Experiments were conducted on a machine powered by 256GB RAM, 32 processors with Intel(R) Xeon(R) Gold 5320 CPU @2.20GHz and a NVIDIA Tesla V100 GPU with 32 GB memory. Each experiment was run 3 times, and the average is reported here. For limited space, some results are reported on specific datasets/models.

Metrics. We used three widely-adopted metrics [23, 96, 98, 99] for local explanations. Each metric was evaluated on m testing vertices u at which $\mathcal{M}$ makes negative predictions. We set $m = 1000$.

(1) *Fidelity*. It measures whether the explanations are faithful to the model's predictions. Since a counterfactual method perturbs the input graph for a prediction switch at u, the counterfactual fidelity is

$$\text{fidelity} = \frac{1}{m}\sum\nolimits_{u} \mathbb{1}(\hat{y}_u \neq y_u),$$

where $\hat{y}_u$ (resp. y_u) is the model prediction on the perturbed $G \ominus \mathbb{O}$ (resp. the original G), and $\mathbb{1}(\cdot)$ returns 1 if $\hat{y}_u \neq y_u$. *i.e.,* it measures the ratio of successful prediction switches over all testing vertices.

(2) *Sparsity*. It measures the fraction of edges perturbed, defined as:

$$\text{sparsity} = \frac{1}{m}\sum_{u} \frac{\#\text{perturb\_edges}_u}{\#\text{total\_edges}_u},$$

where $\#\text{perturb\_edges}_u$ (resp. $\#\text{total\_edges}_u$) is the number of perturbed edges (resp. edges in the L-hop neighborhood graph) of u.

(3) *Feature ratio*. We also report the feature ratio, *i.e.,* the average ratio of attributes perturbed in the L-hop neighborhood graph.

For global explanations, we used another two popular metrics [62, 97]: (1) *overall recognizability* that checks the recognizability of a GNN model $\mathcal{M}$ on global explanations $\mathcal{E}$ (*e.g.,* $\mathcal{E} = \Sigma$ in NEX), *i.e.,*

$$\text{recog} = \frac{1}{|\mathcal{E}|}\sum\nolimits_{\text{exp}\in\mathcal{E}} \mathbb{1}(\hat{y}(\text{exp}) = \text{false}),$$

where exp is an explanation in $\mathcal{E}$ (*e.g.,* a canonical graph formed by φ in Σ), and $\hat{y}(\text{exp})$ is the prediction of $\mathcal{M}$ by feeding exp into $\mathcal{M}$; and (2) *reliability* that gives the coverage of $\mathcal{E}$ on all testing vertices, *i.e.,*

$$\text{relia} = \frac{|\bigcup_{\text{exp}\in\mathcal{E}} \text{testVertices(exp)}|}{\#\text{totalVertices}},$$

where #totalVertices and testVertices (exp) are the total number of vertices and the set of vertices that can be explained by exp (*e.g.,* a vertex v can be explained by φ if φ is applicable at v), respectively.

Experimental results. We next report our findings.

Exp-1: Local effectiveness. We first tested effect of CFE in NEX.

Effectiveness. In Table 2, on average, the fidelity of NEX is 8.30%, 60.85%, 53.71%, 24.98% and 20.17% higher than GVEX, CFGraph, CF-Exp, CLEAR and InduceCE on the four datasets, respectively, up to 16.91%, 105.95%, 92.38%, 43.19% and 45.29%, since NEX finds necessary updates based on cost-effectiveness while the baselines depend heavily on hyper-parameters. Although GVEX also achieves reasonable fidelity, it is far less efficient (see Exp-2). The sparsity of NEX outperforms all baselines by 3.25× on average, up to 7.00×.

For feature ratios (Figure 10(a)), NEX perturbs 1.84% features on average to swap predictions. In contrast, the five baselines either do not consider features or apply the same feature mask to all vertices, treating features selected on different vertices the same.

Varying β. We varied β that controls the perturbation (*i.e.,* modify features or remove edges), from 0.001 to 1 in Figure 10(b). When β gets larger, NEX tends to invalidate C_φ by modifying features only, leading to a lower sparsity and a higher feature ratio, as expected. In contrast, the fidelity of NEX slightly fluctuates and achieves the highest when $\beta = 0.01$ (not shown), where it strikes a balance.

Exp-2: Local efficiency. We evaluated the efficiency (*i.e.,* the average time for each testing vertex) of counterfactual explanations.

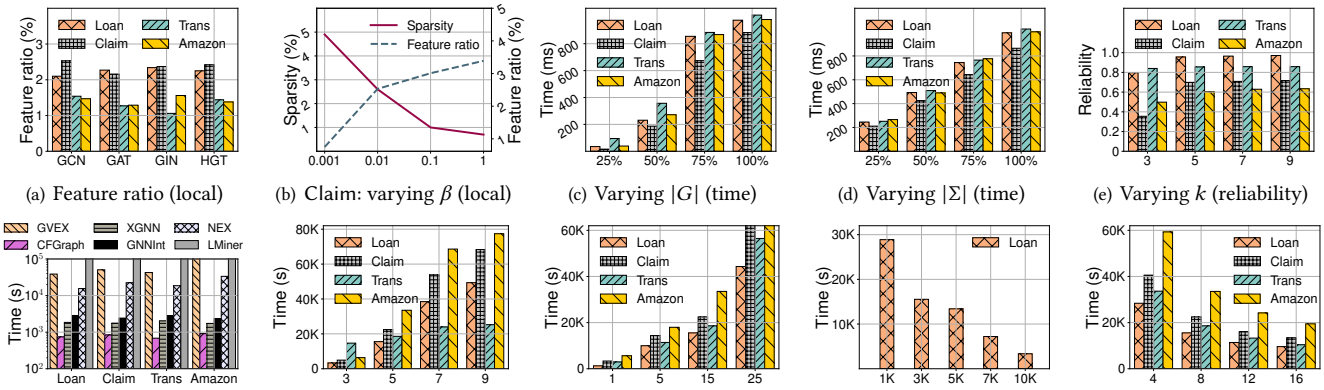
Efficiency. In Table 2, GVEX is slow since it computes Jacobian matrices to identify influential subgraphs (rather than just selecting edges like most other methods), while NEX is 7.30× faster than remaining baselines on average, *e.g.,* on Trans, NEX only takes 1.38s on average. This is because it not only utilizes C_φ to avoid exhaustive enumeration, but also effectively finds perturbations by cost-effectiveness.

Varying $|G|$. We varied the scaling factor of G in Figure 10(c). As expected, it takes longer for NEX to handle larger graphs, *e.g.,* on Trans, NEX is 2.84× slower when G changes from from 50% to 100%.

Varying $|\Sigma|$. We varied the size of Σ in Figure 10(d). It takes longer for NEX to process more RxGNNs in Σ, as expected. Nevertheless, NEX is efficient by handling hundreds of RxGNNs in around 1s.

Exp-3: Global effectiveness. We tested the quality of learned Σ.

Effectiveness. In Table 4, the average recog (resp. relia) of NEX is 15.65%, 167.21%, 80.42% and 17.34% (resp. 31.93%, 253.18%, 302.15% and 171.47%) higher than GVEX, CFGraph, XGNN and GNNInt, respectively, since rule discovery is guided by GNNs; it finds discriminative motifs mostly responsible for the predictions. In contrast, the



(a) Feature ratio (local) (b) Claim: varying β (local) (c) Varying $|G|$ (time) (d) Varying $|\Sigma|$ (time) (e) Varying k (reliability)

(f) Global efficiency (g) Varying k (time) (h) Varying $|\mathcal{S}_M|$ (time) (i) Loan: varying σ (time) (j) Varying n (time)

Figure 10: Performance evaluation

baselines may produce patterns that are hardly satisfied in realty.

Varying k. We varied the maximum size k of motifs from 3 to 9. As shown in Figure 10(e), for $k = 3$, the reliability of NEX is 0.355 on Claim, as the motifs identified are not discriminative enough, and it increases with larger k. Optimal performance is observed at $k = 5$.

Exp-4: Global efficiency. We tested the runtime of RxGNNMiner.

Efficiency. As shown in Figure 10(f), NEX is at least 2.25× faster than GVEX (which runs out of time on Amazon), but it is not the fastest method, due to the inherent cost of rule discovery (which, however, is a one-time, offline process), since it identifies both important patterns and features. In contrast, the baselines generate explanations without considering individual attributes. Compared to levelwise LMiner, which fails to finish in 24 hours, NEX mines rules through motif identification and composition. Although Trans is the largest dataset, it is not the slowest dataset for NEX because it has fewer vertex label types (Table 3), yielding simpler motif combinations.

Varying k or $|\mathcal{S}_M|$. We tested the efficiency of NEX by varying k in Figure 10(g). With a larger k, there are more k-bounded motifs, incurring longer runtime as expected, *e.g.,* when k is changed from 3 to 9, the rule discovery of NEX is 10.64× slower on average. Note that compared with other datasets, its time growth trend on Trans is relatively smooth; this is also attributed to the fewer label types where a small k is sufficient to generate most frequent patterns.

The trend of varying #motifs in $\mathcal{S}_M$ is similar (Figure 10(h)).

Varying σ or δ. Varying support σ from 1K to 10K on Loan, we show the time of NEX in Figure 10(i). Similar to prior studies, when σ increases, it runs faster by filtering more rules via anti-monotonicity. However, since confidence is not anti-monotonic, NEX gets slightly slower (not shown) and more rules are checked when δ increases.

Varying n. Varying the number n of processors, we tested the parallel scalability of RxGNNMiner. As shown in Figure 10(j), it is 3.06× faster when n varies from 4 to 16, *i.e.,* it is parallely scalable.

Exp-5: Ablation study. We conducted an ablation study for NEX.

Local explanations. In Figure 9(a), we tested CFE in NEX with (a) a variant noC of CFE without clustering (*i.e.,* compute cost-effectiveness for all vertices) and (b) the brute-force method BF that enumerates all witnesses (Section 5). As shown there, CFE effectively decides the right vertex to remove and encapsulates the search space, improving the efficiency by 1.24× and 3.55×, respectively.

Global explanations. We tested RxGNNMiner with (a) a variant limited to star patterns (*i.e.,* an adaptation of [23]); (b) a version without DQN, exhaustively enumerating the most confident σ-frequent pattern; (c) a version without perplexity predicate ordering (PPO). Here (a) (resp. (b)(c)) is compared for effectiveness (resp. efficiency)

By restricting the graph patterns of RxGNNs to a special case of star patterns [22–24], the expressivity is compromised, *e.g.,* NEX's recog and relia for GCN drop from 0.731 to 0.427 and from 0.604 to 0.224 on Amazon (not shown), respectively. As for efficiency, the use of DQN and PPO speeds up rule discovery by 1.93× and 1.21× on average, respectively, as shown in Figure 9(b), since DQN efficiently guides the composition and PPO optimizes multi-thread parallelism.

Exp-6: Case study. We report a case study on Claim in Figure 11, at a claim u_0 (ID 100003). To illustrate, we only visualize representative perturbations in a simplified 2-hop neighborhood graph G_L of u_0.

As part of global explanation Σ, a rule in Σ is $\varphi_c = Q_c[\bar{x}, x_0](X_c \rightarrow \neg\mathcal{M}_c(x_0))$, where Q_c is in Figure 11(a) and X_c is x_2.numDiseases $\geq 8 \wedge x_4$.isGroupFraud = true $\wedge\ \neg$1WL(x_4) $\wedge\ x_5$.isFraud = true. It says that claim x_0 is predicted as fraudulent by $\mathcal{M}_c$ *because* (a) x_0 has the same diagnosis group and provider as claim x_4 which is part of a coordinated group fraud scheme and is put in the negative class by 1-WL test, (b) the beneficiary x_2 of x_0 has at least 8 diseases, and (c) another claim x_5 of x_2 is already labeled as fraudulent.

Utilizing Σ, NEX modifies critical attributes and edges around u_0 related to fraudulent activities. A counterfactual explanation at u_0 is given in Figure 11(a), *e.g.,* to make φ_c no longer applicable at u_0, one can modify isFraud of the 2-hop neighbor u_1 of u_0 to true. Applying all perturbations resulting from Σ, $\mathcal{M}_c(u_0)$ is swapped to positive.

In comparison, no baseline considers the impact of individual attributes. GVEX finds a factual subgraph G_f from G_L and crudely removes G_f from G_L to swap prediction in the resulting (green) graph. CF-Exp only allows edge removal; even after it removes multiple edges, its prediction switch fails. Clear swaps prediction by generating explanations that violate domain specifications (*e.g.,* there cannot be edges between groups and beneficiaries). InduCE not only removes edges, but also inserts edges to irrelevant beneficiaries to mislead the model (CFGraph is similar to InduCE and thus omitted).

Summary. We find the following. (1) NEX gives effective local counterfactual explanations. Over four real-life graphs, its average fidelity and sparsity are 33.60% and 3.25× better than the prior methods, respectively. (2) NEX is efficient. For each vertex, it takes 1.38s on average to generate a counterfactual explanation on a graph with 15.4M vertices and 12.7M edges. (3) The set Σ of RxGNNs provides accurate global explanations. Its recog and relia for global expla-

GNN	Method	Loan		Claim		Trans		Amazon	
		recog	relia	recog	relia	recog	relia	recog	relia
GCN	GVEX	0.763	0.652	0.843	0.575	0.687	0.672	0.691	0.502
	CFGraph	0.409	0.221	0.276	0.158	0.313	0.496	0.426	0.145
	XGNN	0.373	0.167	0.656	0.174	0.528	0.353	0.418	0.183
	GNNInt	0.755	0.545	0.783	0.218	0.691	0.475	0.697	0.221
	NEX	**1.000**	**0.956**	**0.934**	**0.699**	**0.984**	**0.855**	**0.731**	**0.604**
GAT	GVEX	0.768	0.607	0.857	0.551	0.698	0.688	0.663	0.486
	CFGraph	0.384	0.229	0.255	0.174	0.299	0.51	0.445	0.175
	XGNN	0.414	0.143	0.613	0.176	0.535	0.316	0.47	0.191
	GNNInt	0.735	0.511	0.809	0.113	0.737	0.458	0.719	0.245
	NEX	**1.000**	**0.961**	**0.967**	**0.702**	**0.971**	**0.851**	**0.745**	**0.601**
GIN	GVEX	0.807	0.616	0.833	0.549	0.696	0.654	0.682	0.508
	CFGraph	0.432	0.197	0.273	0.144	0.255	0.539	0.476	0.184
	XGNN	0.410	0.182	0.595	0.154	0.458	0.320	0.459	0.138
	GNNInt	0.787	0.437	0.771	0.166	0.677	0.305	0.688	0.267
	NEX	**1.000**	**0.955**	**0.953**	**0.658**	**1.000**	**0.887**	**0.744**	**0.614**
HGT	GVEX	0.776	0.634	0.861	0.573	0.678	0.694	0.661	0.455
	CFGraph	0.434	0.178	0.212	0.197	0.26	0.474	0.424	0.177
	XGNN	0.398	0.192	0.587	0.160	0.503	0.298	0.429	0.180
	GNNInt	0.765	0.507	0.797	0.188	0.662	0.437	0.694	0.233
	NEX	**1.000**	**0.944**	**0.932**	**0.691**	**0.965**	**0.872**	**0.728**	**0.617**

Table 4: Global effectiveness

nation are 70.16% and 189.68% higher than the prior methods on average, up to 339.62% and 572.03%, respectively. (4) Its discovery algorithm is parallelly scalable, and is able to learn rules from large G.

7 RELATED WORK

Explanation methods. Prior methods are categorized as follows.

(1) Local explanation. Explanation methods for GNN-based models can be divided into three classes: (a) *Factual methods* [11, 37, 46, 48, 52, 60, 67, 69, 77, 78, 86, 96, 99] by finding the most important sub-structures or features for GNNs to make a prediction. (b) *Counterfactual methods* by finding the smallest updates in the input graph that change the prediction of GNNs [8–10, 20, 59, 63, 70, 85] (surveyed in [68]), *e.g.,* CF-GNNExplainer [59] finds the optimal perturbations to swap a prediction; Clear [63] generates counterfactual graphs by a graph variational autoencoder; InduCE [85] employs reinforcement learning on a combinatorial optimization problem; and LocalCE [70] searches a pair of similar but differently classified vertices as a counterfactual evidence. (c) *FCF methods* that provide both factual and counterfactual explanations [17, 71, 72, 80].

(2) Global explanation. Global explanation methods include XGNN (reinforcement learning) [97], GNNInterpreter (numerical optimization) [88], DAG (randomized greedy) [62], and GLGExplainer (prototype learning) [88]. Besides, GCFExplainer [47] focuses on global counterfactual reasoning by finding candidate counterfactual graphs via vertex-reinforced random walks. ComRecGC [36] generates global counterfactual graphs relevant to all input graphs.

Although some methods (*e.g.,* XGNN) do not take specific graphs as input, they also rely on pre-trained GNNs and thus implicitly depend on specific training data. NEX generates both local and global explanations and hence naturally takes a specific graph as input.

(3) Both global and local explanation. GVEX [16] designs a two-tier explanation structure that consists of global graph patterns and local explanation subgraphs. CFGraph [5] finds counterfactual explanations by adding or removing edges and aggregating the result as global explanations. The most related work, Makex [23], provides rule-based explanations for GNN link predictions but uses less expressive rules with limited patterns and predicates. It supports only factual explanations to identify sufficient substructures, without requiring negation or graph edits. In contrast, counterfactual explanations demand graph modifications as necessary conditions, which require rules with negated predicates (see[4] for details).

(4) Explanations on non-graph data. To explain blackbox models, rules-based methods [14, 39, 75] have been studied on tabular data. There have been systems for identifying plausible and feasible feature changes that swap ML predictions, *e.g.,* Geco [76], FACET [83], Mace [53] and Dice [66]. However, it remains challenging to explain GNN predictions since GNN models involve topology on graphs.

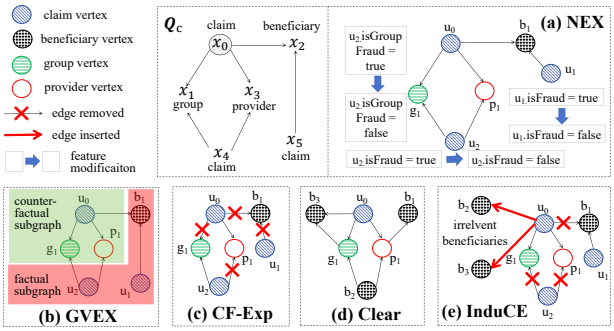


Figure 11: Case study on Claim

We differ from prior works in some key aspects: (a) RxGNNs use both graph patterns and preconditions to reveal not just what structures are decisive, but also under what conditions predictions are made, beyond simple meta-paths or subgraphs used in earlier work; (b) we focus on counterfactual explanations that flip predictions, unlike factual ones that require no graph changes; (c) we provide a unified framework for global and local explanations for any GNN classifier; and (d) unlike rules in [14, 39, 75], which offer only local explanations using constant predicates, RxGNNs support a full range of predicates, including negation, enabling expressive reasoning.

Rules for graphs. Rules have been developed for graphs, for (a) cleaning, *e.g.,* GFDs [35], GEDs [32] and GCRs [24], (b) association analyses, *e.g.,* GPARs [34], GARs [30] and TACOs [31], (c) improving ML recommendation, *e.g.,* TIEs [22], and (d) factual explanation, *e.g.,* REPs [23]. Rule discovery/learning algorithms have been studied for GFDs [29], GARs [25], TACOs [31], TIEs [22] and REPs [23].

Unlike prior rules, RxGNNs (1) take $\neg\mathcal{M}$ as their consequences, (2) support general patterns, beyond star patterns in [22–24], and (3) support a full range of predicates, *e.g.,* negated and 1-WL predicates, to capture negative/diverse behaviors of GNNs. An expressivity comparison for both graph rules and tabular rules is provided in [4], which summarizes various rule properties. Besides, the discovery of RxGNNs is particularly guided by the prediction signals of GNNs.

Graph pattern mining. Various techniques have been studied to discover frequent structures in graph data [21, 45, 50, 57, 73, 82, 94, 95] (surveyed in [51]). However, graph mining alone does not suffices to explain GNNs. We adopt gSpan [95] for its widespread use; advanced mining algorithms can also be readily plugged in.

8 CONCLUSION

NEX is the first to address GNN-based classifiers $\mathcal{M}$ by (a) providing both global and local explanations of negative $\mathcal{M}$ predictions in a logical framework; (b) proposing RxGNNs with negated predicates and 1-WL test that capture diverse behaviors of $\mathcal{M}$, more expressive than prior graph rules; (c) developing a model-guided and parallelly scalable algorithm for discovering RxGNNs as global explanations; and (d) providing a cost-effective algorithm for computing local counterfactual explanations, with vertex-filtering techniques. Empirical results confirm that NEX is promising in practice.

Future work includes extending NEX to explain non-GNN models, and provide both factual and counterfactual explanations.

REFERENCES

[1] 2024. The Dataset for Fraudulent Activities Detection. *https://www.kaggle.com/datasets/rohit265/fraud-detection-dynamics-financial-transaction.*

[2] 2024. The Dataset for Loan Approval Prediction. *https://www.kaggle.com/datasets/hydracsnova/loan-approval-dataset.*

[3] 2025. Amazon product graph. *https://cseweb.ucsd.edu/~jmcauley/datasets/.*

[4] 2025. Code, datasets and full version. https://github.com/hdqlkncso24jvf/NEX.

[5] Carlo Abrate and Francesco Bonchi. 2021. Counterfactual graphs for explainable classification of brain networks. In *SIGKDD*. ACM New York, NY, USA, 2495–2504.

[6] Waseem Akhtar, Alvaro Cortés-Calabuig, and Jan Paredaens. 2010. Constraints in RDF. In *SDKB*.

[7] Noga Alon, Raphael Yuster, and Uri Zwick. 1995. Color-coding. *Journal of the ACM (JACM)* 42, 4 (1995), 844–856.

[8] Shuai An and Yang Cao. 2024. Counterfactual Explanation at Will, with Zero Privacy Leakage. *PACMMOD* (2024).

[9] Hervé-Madelein Attolou, Katerina Tzompanaki, Kostas Stefanidis, and Dimitris Kotzinos. 2024. Why-not explainable graph recommender. In *ICDE*.

[10] Mohit Bajaj, Lingyang Chu, Zi Yu Xue, Jian Pei, Lanjun Wang, Peter Cho-Ho Lam, and Yong Zhang. 2021. Robust counterfactual explanations on graph neural networks. *Advances in Neural Information Processing Systems* 34 (2021), 5644–5655.

[11] Federico Baldassarre and Hossein Azizpour. 2019. Explainability Techniques for Graph Convolutional Networks. In *ICML Workshop "Learning and Reasoning with Graph-Structured Representations"*.

[12] L Breiman, JH Friedman, R Olshen, and CJ Stone. 1984. Classification and Regression Trees. (1984).

[13] Jin-yi Cai, Martin Fürer, and Neil Immerman. 1992. An optimal lower bound on the number of variables for graph identification. *Comb.* 12, 4 (1992), 389–410.

[14] Chaofan Chen, Kangcheng Lin, Cynthia Rudin, Yaron Shaposhnik, Sijia Wang, and Tong Wang. 2018. An Interpretable Model with Globally Consistent Explanations for Credit Risk. *CoRR* abs/1811.12615 (2018). arXiv:1811.12615 http://arxiv.org/abs/1811.12615

[15] Haoxian Chen, Wenfei Fan, and Jiaye Zheng. 2025. Incremental Rule Discovery in Response to Parameter Updates. *Proc. ACM Manag. Data* 3, 3 (2025), 175:1–175:28.

[16] Tingyang Chen, Dazhuo Qiu, Yinghui Wu, Arijit Khan, Xiangyu Ke, and Yunjun Gao. 2024. View-based explanations for graph neural networks. *Proc. ACM Manag. Data* 2, 1 (2024), 1–27.

[17] Ziheng Chen, Fabrizio Silvestri, Jia Wang, Yongfeng Zhang, Zhenhua Huang, Hongshik Ahn, and Gabriele Tolomei. 2022. GREASE: Generate Factual and Counterfactual Explanations for GNN-based Recommendations. *CoRR* abs/2208.04222 (2022).

[18] Xu Chu, Ihab F. Ilyas, and Paolo Papotti. 2013. Discovering Denial Constraints. *PVLDB* 6, 13 (2013), 1498–1509.

[19] Alvaro Cortés-Calabuig and Jan Paredaens. 2012. Semantics of Constraints in RDFS. In *AMW*.

[20] Jiajun Cui, Minghe Yu, Bo Jiang, Aimin Zhou, Jianyong Wang, and Wei Zhang. 2024. Interpretable Knowledge Tracing via Response Influence-based Counterfactual Reasoning. In *ICDE*.

[21] Mohammed Elseidy, Ehab Abdelhamid, Spiros Skiadopoulos, and Panos Kalnis. 2014. GRAMI: Frequent Subgraph and Pattern Mining in a Single Large Graph. *PVLDB* 7, 7 (2014), 517–528.

[22] Lihang Fan, Wenfei Fan, Ping Lu, Chao Tian, and Qiang Yin. 2024. Enriching Recommendation Models with Logic Conditions. *Proc. ACM Manag. Data* 1, 3 (2024), 210:1–210:28.

[23] Wenfei Fan, Lihang Fan, Dandan Lin, and Min Xie. 2025. Explaining GNN-based Recommendations in Logic. *PVLDB* (2025).

[24] Wenfei Fan, Wenzhi Fu, Ruochun Jin, Muyang Liu, Ping Lu, and Chao Tian. 2023. Making It Tractable to Catch Duplicates and Conflicts in Graphs. *Proc. ACM Manag. Data* 1, 1 (2023), 86:1–86:28.

[25] Wenfei Fan, Wenzhi Fu, Ruochun Jin, Ping Lu, and Chao Tian. 2022. Discovering Association Rules from Big Graphs. *PVLDB* 15, 7 (2022), 1479–1492.

[26] Wenfei Fan, Hong Gao, Xibei Jia, Jianzhong Li, and Shuai Ma. 2011. Dynamic constraints for record matching. *VLDB J.* 20, 4 (2011), 495–520.

[27] Wenfei Fan, Floris Geerts, Xibei Jia, and Anastasios Kementsietsidis. 2008. Conditional functional dependencies for capturing data inconsistencies. *TODS* 33, 2 (2008), 6.

[28] Wenfei Fan, Chunming Hu, Xueli Liu, and Ping Lu. 2018. Discovering Graph Functional Dependencies. In *SIGMOD*. 427–439.

[29] Wenfei Fan, Chunming Hu, Xueli Liu, and Ping Lu. 2020. Discovering graph functional dependencies. *ACM Transactions on Database Systems (TODS)* 45, 3 (2020), 1–42.

[30] Wenfei Fan, Ruochun Jin, Muyang Liu, Ping Lu, Chao Tian, and Jingren Zhou. 2020. Capturing Associations in Graphs. *PVLDB* 13, 11 (2020), 1863–1876.

[31] Wenfei Fan, Ruochun Jin, Ping Lu, Chao Tian, and Ruiqi Xu. 2022. Towards Event Prediction in Temporal Graphs. *PVLDB* 15, 9 (2022), 1861–1874.

[32] Wenfei Fan and Ping Lu. 2019. Dependencies for Graphs. *ACM Trans. Database Syst. (TODS)* 44, 2 (2019), 5:1–5:40.

[33] Wenfei Fan, Ping Lu, and Chao Tian. 2020. Unifying Logic Rules and Machine Learning for Entity Enhancing. *Sci. China Inf. Sci.* 63, 7 (2020).

[34] Wenfei Fan, Xin Wang, Yinghui Wu, and Jingbo Xu. 2015. Association Rules with Graph Patterns. *PVLDB* 8, 12 (2015), 1502–1513.

[35] Wenfei Fan, Yinghui Wu, and Jingbo Xu. 2016. Functional dependencies for graphs. In *SIGMOD*. ACM, 1843–1857.

[36] Gregoire Fournier and Sourav Medya. 2025. COMRECGC: Global Graph Counterfactual Explainer through Common Recourse. *arXiv preprint arXiv:2505.07081* (2025).

[37] Thorben Funke, Megha Khosla, Mandeep Rathee, and Avishek Anand. 2022. Z orro: Valid, sparse, and stable explanations in graph neural networks. *IEEE Transactions on Knowledge and Data Engineering* (2022).

[38] Michael Garey and David Johnson. 1979. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman and Company.

[39] Zixuan Geng, Maximilian Schleich, and Dan Suciu. 2022. Computing Rule-Based Explanations by Leveraging Counterfactuals. *PVLDB* 16, 3 (2022), 420–432.

[40] Martin Grohe. 2020. word2vec, node2vec, graph2vec, X2vec: Towards a Theory of Vector Embeddings of Structured Data. In *PODS*. ACM, 1–16.

[41] Martin Grohe. 2021. The Logic of Graph Neural Networks. In *LICS*. 1–17.

[42] Mandlenkosi Victor Gwetu, Jules-Raymond Tapamo, and Serestina Viriri. 2019. Random Forests with a Steepend Gini-Index Split Function and Feature Coherence Injection. In *MLN*. 255–272.

[43] Myoungji Han, Hyunjoon Kim, Geonmo Gu, Kunsoo Park, and Wook-Shin Han. 2019. Efficient subgraph matching: Harmonizing dynamic programming, adaptive matching order, and failing set together. In *SIGMOD*. 1429–1446.

[44] Ziniu Hu, Yuxiao Dong, Kuansan Wang, and Yizhou Sun. 2020. Heterogeneous graph transformer. In *The Web conference 2020*. 2704–2710.

[45] Jun Huan, Wei Wang, Jan Prins, and Jiong Yang. 2004. Spin: Mining maximal frequent subgraphs from graph databases. In *SIGKDD*.

[46] Qiang Huang, Makoto Yamada, Yuan Tian, Dinesh Singh, and Yi Chang. 2022. GraphLIME: Local Interpretable Model Explanations for Graph Neural Networks. *TKDE* 35, 7 (2022), 6968–6972.

[47] Zexi Huang, Mert Kosan, Sourav Medya, Sayan Ranu, and Ambuj Singh. 2023. Global counterfactual explainer for graph neural networks. In *ACM International Conference on Web Search and Data Mining*. 141–149.

[48] Zhenhua Huang, Kunhao Li, Shaojie Wang, Zhaohong Jia, Wentao Zhu, and Sharad Mehrotra. 2024. SES: Bridging the Gap Between Explainability and Prediction of Graph Neural Networks. In *ICDE*.

[49] Ykä Huhtala, Juha Kärkkäinen, Pasi Porkka, and Hannu Toivonen. 1999. TANE: An efficient algorithm for discovering functional and approximate dependencies. *The computer journal* 42, 2 (1999), 100–111.

[50] Anand Padmanabha Iyer, Zaoxing Liu, Xin Jin, Shivaram Venkataraman, Vladimir Braverman, and Ion Stoica. 2018. {ASAP}: Fast, approximate graph pattern mining at scale. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 745–761.

[51] Chuntao Jiang, Frans Coenen, and Michele Zito. 2013. A survey of frequent subgraph mining algorithms. *Knowledge Eng. Review* 28, 01 (2013), 75–105.

[52] Jaykumar Kakkad, Jaspal Jannu, Kartik Sharma, Charu Aggarwal, and Sourav Medya. 2023. A survey on explainability of graph neural networks. *arXiv preprint arXiv:2306.01958* (2023).

[53] Amir-Hossein Karimi, Gilles Barthe, Borja Balle, and Isabel Valera. 2019. Model-Agnostic Counterfactual Explanations for Consequential Decisions. *CoRR* abs/1905.11190 (2019).

[54] Elias Khalil, Hanjun Dai, Yuyu Zhang, Bistra Dilkina, and Le Song. 2017. Learning combinatorial optimization algorithms over graphs. *Advances in neural information processing systems* 30 (2017).

[55] Thomas N Kipf and Max Welling. 2017. Semi-supervised classification with graph convolutional networks. *ICLR* (2017).

[56] Clyde P. Kruskal, Larry Rudolph, and Marc Snir. 1990. A Complexity Theory of Efficient Parallel Algorithms. *Theor. Comput. Sci.* 71, 1 (1990), 95–132.

[57] Wenqing Lin, Xiaokui Xiao, and Gabriel Ghinita. 2014. Large-scale frequent subgraph mining in MapReduce. In *ICDE*.

[58] Yunchao Lance Liu, Yu Wang, Oanh Vu, Rocco Moretti, Bobby Bodenheimer, Jens Meiler, and Tyler Derr. 2023. Interpretable chirality-aware graph neural network for quantitative structure activity relationship modeling in drug discovery. In *The AAAI Conference on Artificial Intelligence*, Vol. 37. 14356–14364.

[59] Ana Lucic, Maartje A Ter Hoeve, Gabriele Tolomei, Maarten De Rijke, and Fabrizio Silvestri. 2022. CF-GNNExplainer: Counterfactual Explanations for Graph Neural Networks. In *International Conference on Artificial Intelligence and Statistics*. PMLR, 4499–4511.

[60] Dongsheng Luo, Wei Cheng, Dongkuan Xu, Wenchao Yu, Bo Zong, Haifeng Chen, and Xiang Zhang. 2020. Parameterized explainer for graph neural network. *NeurIPS* 33 (2020), 19620–19631.

[61] Yice Luo, Guannan Wang, Yongchao Liu, Jiaxin Yue, Weihong Cheng, and Binjie Fei. 2023. FAF: A Risk Detection Framework on Industry-Scale Graphs.

In *ACM International Conference on Information and Knowledge Management*. 4717–4723.

[62] Ge Lv and Lei Chen. 2023. On Data-Aware Global Explainability of Graph Neural Networks. *PVLDB* 16, 11 (2023), 3447–3460.

[63] Jing Ma, Ruocheng Guo, Saumitra Mishra, Aidong Zhang, and Jundong Li. 2022. Clear: Generative counterfactual explanations on graphs. *Advances in neural information processing systems* 35 (2022), 25895–25907.

[64] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin A. Riedmiller, Andreas Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. 2015. Human-level control through deep reinforcement learning. *Nat.* (2015).

[65] H. L. Morgan. 2065. The Generation of a Unique Machine Description for Chemical Structures-A Technique Developed at Chemical Abstracts Service. *Journal of Chemical Documentation* 5, 2 (2065), 107–113.

[66] Ramaravind Kommiya Mothilal, Amit Sharma, and Chenhao Tan. 2020. Explaining machine learning classifiers through diverse counterfactual explanations. In *Conference on Fairness, Accountability, and Transparency (FAT)*. ACM, 607–617.

[67] Phillip E Pope, Soheil Kolouri, Mohammad Rostami, Charles E Martin, and Heiko Hoffmann. 2019. Explainability methods for graph convolutional neural networks. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 10772–10781.

[68] Mario Alfonso Prado-Romero, Bardh Prenkaj, Giovanni Stilo, and Fosca Giannotti. 2024. A survey on graph counterfactual explanations: Definitions, methods, evaluation, and research challenges. *SIGMOD* (2024).

[69] Dazhuo Qiu, Haolai Che, Arijit Khan, and Yinghui Wu. 2025. Generating Skyline Explanations for Graph Neural Networks. *arXiv preprint arXiv:2505.07635* (2025).

[70] Dazhuo Qiu, Jinwen Chen, Arijit Khan, Yan Zhao, and Francesco Bonchi. 2025. Finding Counterfactual Evidences for Node Classification. *arXiv preprint arXiv:2505.11396* (2025).

[71] Dazhuo Qiu, Mengying Wang, Arijit Khan, and Yinghui Wu. 2024. Generating Robust Counterfactual Witnesses for Graph Neural Networks. *ICDE* (2024).

[72] Dazhuo Qiu, Mengying Wang, Arijit Khan, and Yinghui Wu. 2024. Generating robust counterfactual witnesses for graph neural networks. In *ICDE*. IEEE, 3351–3363.

[73] Sayan Ranu and Ambuj K Singh. 2009. GraphSig: A scalable approach to mining significant subgraphs in large graph databases. In *ICDE*. IEEE, 844–855.

[74] 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. *PVLDB* 15, 3 (2021), 427–436.

[75] Cynthia Rudin and Yaron Shaposhnik. 2023. Globally-Consistent Rule-Based Summary-Explanations for Machine Learning Models: Application to Credit-Risk Evaluation. *J. Mach. Learn. Res.* 24 (2023), 16:1–16:44.

[76] Maximilian Schleich, Zixuan Geng, Yihong Zhang, and Dan Suciu. 2021. GeCo: Quality Counterfactual Explanations in Real Time. *PVLDB* 14, 9 (2021), 1681–1693.

[77] Michael Sejr Schlichtkrull, Nicola De Cao, and Ivan Titov. 2020. Interpreting Graph Neural Networks for NLP With Differentiable Edge Masking. In *ICLR*.

[78] Caihua Shan, Yifei Shen, Yao Zhang, Xiang Li, and Dongsheng Li. 2021. Reinforcement learning enhanced explainer for graph neural networks. *Advances in Neural Information Processing Systems* 34 (2021), 22523–22533.

[79] Mengying Sun, Sendong Zhao, Coryandar Gilvary, Olivier Elemento, Jiayu Zhou, and Fei Wang. 2020. Graph convolutional networks for computational drug development and discovery. *Briefings in bioinformatics* 21, 3 (2020), 919–935.

[80] Juntao Tan, Shijie Geng, Zuohui Fu, Yingqiang Ge, Shuyuan Xu, Yunqi Li, and Yongfeng Zhang. 2022. Learning and evaluating graph neural network explanations based on counterfactual and factual reasoning. In *The Web Conference*. 1018–1027.

[81] Hao Tang, Cheng Wang, Jianguo Zheng, and Changjun Jiang. 2024. Enabling Graph Neural Networks for Semi-Supervised Risk Prediction in Online Credit Loan Services. *ACM Transactions on Intelligent Systems and Technology* 15, 1 (2024), 1–24.

[82] Marisa Thoma, Hong Cheng, Arthur Gretton, Jiawei Han, Hans-Peter Kriegel, Alex Smola, Le Song, Philip S Yu, Xifeng Yan, and Karsten M Borgwardt. 2010. Discriminative frequent subgraph mining with optimality guarantees. *Statistical Analysis and Data Mining: The ASA Data Science Journal* 3, 5 (2010), 302–318.

[83] Peter M. VanNostrand, Huayi Zhang, Dennis M. Hofmann, and Elke A. Rundensteiner. 2023. FACET: Robust Counterfactual Explanation Analytics. *Proc. ACM Manag. Data* 1, 4 (2023), 242:1–242:27.

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

[85] Samidha Verma, Burouj Armgaan, Sourav Medya, and Sayan Ranu. 2024. InduCE: Inductive Counterfactual Explanations for Graph Neural Networks. *Transactions on Machine Learning Research* (2024).

[86] Minh Vu and My T Thai. 2020. PGM-explainer: Probabilistic graphical model explanations for graph neural networks. *NeurIPS* 33 (2020), 12225–12235.

[87] Sandra Wachter, Brent D. Mittelstadt, and Chris Russell. 2017. Counterfactual Explanations without Opening the Black Box: Automated Decisions and the GDPR. *CoRR* abs/1711.00399 (2017). arXiv:1711.00399 http://arxiv.org/abs/1711.00399

[88] Xiaoqi Wang and Han Wei Shen. 2022. GNNInterpreter: A Probabilistic Generative Model-Level Explanation for Graph Neural Networks. In *International Conference on Learning Representations*.

[89] B. Yu. Weisfieler and A. A. Leman. 1968. The reduction of a graph to canonical form and the algebra which appears therein. *NIT* 2 (1968).

[90] Catharine Wyss, Chris Giannella, and Edward Robertson. 2001. FastFDs: A heuristic-driven, depth-first algorithm for mining functional dependencies from relation instances extended abstract. In *DaWaK*.

[91] 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.

[92] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. 2019. How Powerful are Graph Neural Networks?. In *ICLR*.

[93] Keyulu Xu, Chengtao Li, Yonglong Tian, Tomohiro Sonobe, Ken-ichi Kawarabayashi, and Stefanie Jegelka. 2018. Representation learning on graphs with jumping knowledge networks. In *International conference on machine learning*. PMLR, 5453–5462.

[94] Xifeng Yan, Hong Cheng, Jiawei Han, and Philip S Yu. 2008. Mining significant graph patterns by leap search. In *SIGMOD*. 433–444.

[95] Xifeng Yan and Jiawei Han. 2002. gSpan: Graph-based substructure pattern mining. In *IEEE International Conference on Data Mining*. IEEE, 721–724.

[96] Zhitao Ying, Dylan Bourgeois, Jiaxuan You, Marinka Zitnik, and Jure Leskovec. 2019. GNNExplainer: Generating Explanations for Graph Neural Networks. In *NeurIPS*. 9240–9251.

[97] Hao Yuan, Jiliang Tang, Xia Hu, and Shuiwang Ji. 2020. XGNN: Towards model-level explanations of graph neural networks. In *SIGKDD*. ACM, 430–438.

[98] 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* 45, 5 (2022), 5782–5799.

[99] Hao Yuan, Haiyang Yu, Jie Wang, Kang Li, and Shuiwang Ji. 2021. On explainability of graph neural networks via subgraph explorations. In *ICML*. PMLR, 12241–12252.

[100] Seongjun Yun, Minbyul Jeong, Raehyun Kim, Jaewoo Kang, and Hyunwoo J Kim. 2019. Graph transformer networks. *Advances in neural information processing systems* 32 (2019).

[101] Rui Zhang, Dawei Cheng, Jie Yang, Yi Ouyang, Xian Wu, Yefeng Zheng, and Changjun Jiang. 2024. Pre-trained online contrastive learning for insurance fraud detection. In *The AAAI Conference on Artificial Intelligence*, Vol. 38. 22511–22519.

[102] Shuai Zhang, Hongkang Li, Meng Wang, Miao Liu, Pin-Yu Chen, Songtao Lu, Sijia Liu, Keerthiram Murugesan, and Subhajit Chaudhury. 2023. On the convergence and sample complexity analysis of deep q-networks with ϵ-greedy exploration. *Advances in Neural Information Processing Systems* 36 (2023), 13064–13102.

[103] Markus Zopf. 2022. 1-WL expressiveness is (almost) all you need. In *International Joint Conference on Neural Networks (IJCNN)*. IEEE, 1–8.

APPENDIX

Appendix A: Details of Deep Q-learning

Note that motif composition can lead to a large state space. To this end, we adopt deep Q-learning (DQN) [64] to learn the policy since DQN is good at handling large state spaces and modeling complex structures with neural networks for better generalizability. While traditional Q-learning is simpler, it struggles when handling large state spaces and can be inefficient due to large Q-tables.

Below we first formulate the composition as a Markov Decision Process (MDP) and then present the training and inference of DQN.

MDP. Given a set S of motifs, we create an initial pattern $Q_0[\bar{x}, x_0]$ by merging vertices x_0 from all motifs in S into one x_0 in Q_0. We model the composition as a sequential decision process: given the current $Q[\bar{x}, x_0]$, we make a decision by selecting a pair of vertices in Q with the same label to merge. This can be modeled as an MDP.

- *State*: The current pattern Q represents a state s.
- *Action*: Each mergable pair (v, v') of vertices in Q (*i.e.,* $L_Q(v) = L_Q(v')$) is an action a. We also use a special action [end] to terminate the composition, and stop if the $\text{supp}(Q, G) < \sigma$ (by anti-monotonicity). Denote the action space of a state s by $\mathcal{A}(s)$.
- *Transition*: We transit from a state to another by taking an action, *e.g.,* given the current state $s_i = Q_i$ and an action $a = (v, v')$, we transit to a new state $s_{i+1} = Q_{i+1}$, denoted as $s_i \rightarrow s_{i+1}$, where v and v' are represented as a single merged vertex in Q_{i+1}.
- *Reward*: A reward r is associated with the transition $s_i \rightarrow s_{i+1}$, indicating the effect of taking action (v, v'). Specifically,

$$r = \begin{cases} -1 & \text{if } \text{supp}(Q_{i+1}, G) < \sigma, \\ \text{conf}(Q_{i+1}, G) - \text{conf}(Q_i, G) & \text{if } \text{supp}(Q_{i+1}, G) \geq \sigma. \end{cases}$$

Training of DQN. To utilize DQN for pattern composition, we adopt several component networks: (a) a *state encoder*, $\text{encoder}_\phi(Q)$ (resp. *action encoder*, $\text{encoder}_\phi(v, v')$), with parameter ϕ that maps a state $s = Q$ (resp. an action $a = (v, v')$)) to an embedding $\mathbf{e}_s$ (resp. $\mathbf{e}_a$), and (b) a *Q-value network* $\mathbb{Q}_\theta(s, a)$ with parameter θ that predicts the expected accumulated reward (*i.e.,* Q-value) by taking action a at s.

More specifically, we leverage the vertex embedding $\mathbf{e}_v^L$ outputted by the L-layer GNN model $\mathcal{M}$ to encode a pattern Q, *i.e.,*

$$\mathbf{e}_s = \text{encoder}_\phi(s) = \text{readout}(\{\mathbf{e}_v^L \mid v \in Q \text{ where } s = Q\}),$$

where the readout function aggregates individual vertex embeddings into a pattern-level representation. Similarly, we concatenate the vertex embeddings to get the action embedding, *i.e.,*

$$\mathbf{e}_a = \text{encoder}_\phi(a) = \text{concat}(\mathbf{e}_v^L, \mathbf{e}_{v'}^L), \text{ where } a = (v, v').$$

For the Q-value network, we adopt $\mathbb{Q}_\theta(s, a) = \text{FFN}_\theta(\mathbf{e}_s, \mathbf{e}_a)$, where FFN() is a feed-forward neural network. We train the network with the ϵ-greedy strategy, minimizing the loss $\mathcal{L}(\theta, \phi)$:

$$\mathbb{E}_{(s_i, a, r, s_{i+1}) \sim \mathcal{T}} \left[\left(r + \gamma \max_{a' \in \mathcal{A}(s_{i+1})} \mathbb{Q}_{\theta'}(s_{i+1}, a') - \mathbb{Q}_\theta(s_i, a) \right)^2 \right],$$

where $\mathcal{T}$ is a replay buffer containing transitions (s_i, a, r, s_{i+1}) (*i.e.,* we transit from s_i to s_{i+1} by taking action a with reward r), γ is a discount factor, and θ' is the parameter of a target network $\mathbb{Q}_{\theta'}$, periodically copied from $\mathbb{Q}_\theta$ for providing a stable target during training.

Inference of DQN. Given the trained $\mathbb{Q}_\theta(s, a)$, we start with the initial pattern Q_0, and iteratively merge the pair of vertices with the maximum Q-value, until the termination condition is satisfied.

Sample complexity. The sample complexity of DQN with ϵ-greedy is polynomially with the network size (*e.g.,* linear to the dimension of the feature space for state-action pairs and the number of layers in the neural network), and inversely with the square of the desired accuracy. We refer interested readers to more details in [102].

Appendix B: Extended Discussion of Discovery

This section presents extended discussion on RxGNNMiner.

Parallel scalability. We adopt *parallel scalability* [56] to measure the effectiveness of RxGNNMiner. Consider a problem $\mathbb{P}$ on graph G. We denote by $T_s(|I_\mathbb{P}|, |G|)$ the worst-case complexity of a sequential algorithm $\mathcal{A}$ for handling an instance $I_\mathbb{P}$ of $\mathbb{P}$ over G. For a parallel algorithm $\mathcal{A}_p$ for $\mathbb{P}$, we denote by $T_p(|I_\mathbb{P}|, |G|, n)$ its time taken for processing problem instance $I_\mathbb{P}$ on G using n processors. We say that algorithm $\mathcal{A}_p$ is *parallelly scalable relative to* $\mathcal{A}$ if

$$T_p(|I_\mathbb{P}|, |G|, n) = O\big(T_s(|I_\mathbb{P}|, |G|)/n\big)$$

for any instance $I_\mathbb{P}$. That is, the parallel algorithm $\mathcal{A}_p$ is able to "linearly" reduce the sequential cost of a yardstick algorithm $\mathcal{A}$. Denote by SeqMiner the sequential version of RxGNNMiner that validates each RxGNN one by one, without multi-thread parallelism.

Theorem 3: RxGNNMiner *is parallelly scalable to* SeqMiner. □

Proof. We show that the complexity of algorithm RxGNNMiner is $O(\frac{1}{n} \times |\mathcal{S}_\mathcal{M}|^K \sum_{X \in \text{pow}(\mathcal{P})} |G|^{|X|})$ with n processors, where K is the maximum number of motifs in $\mathcal{S}_\mathcal{M}$ that can compose a σ-frequent pattern Q (typically, $K \ll |\mathcal{S}_\mathcal{M}|$), $\mathcal{P}$ is the set of all potential predicates defined on a pattern Q and $\text{pow}(\mathcal{P})$ is the power set of $\mathcal{P}$. This is because (a) the computation is dominated by the validation of support and confidence, where the computational tasks are evenly distributed across n processors, and (b) it examines $\text{pow}(\mathcal{P})$ to generate preconditions X for each pattern Q at worst. □

Space cost. The space is dominated by the storage of minimal RxGNNs, which is $O(|\mathcal{S}_\mathcal{M}|^K \times (N + \#\text{max}|\mathcal{P}|) + |G|)$, where N is the maximum number of vertices for each pattern Q and $\#\text{max} = \binom{|\mathcal{P}|}{\lfloor \frac{|\mathcal{P}|}{2} \rfloor}$ is the maximum number of minimal RxGNNs with the same Q.

Remark on k-bounded motifs. We focus on k-bounded motifs, where the value of k balances cost and discriminability. In practice, users can often (a) start with a conservative k (*e.g.,* $k = 3$), incrementally increase it, and stop when when marginal gains diminish; or (b) set time or memory budget, and dynamically adjust k to stay within resource limits; or (c) leverage data statistics to select a suitable k; or (d) train ML models to guide the choice of k. In addition, an incremental tuning strategy [15] can also be employed for k.

Appendix C: Plausibility and Feasibility

To ensure the plausibility and feasibility of feature modifications [76], we reuse the perplexity-based computation in Section 4.3, to leverage the capability and rich prior knowledge of LLMs. More specifically, for each attribute A, we pre-compute a function ρ_A: $\text{dom}(A) \times \text{dom}(A) \rightarrow \mathbb{R}^+$ for measuring the plausibility and feasibility of a modification, where $\text{dom}(A)$ is the domain of A:

$$\rho_A(a, a') = \exp\left(-\log P_{\text{LLM}}(\text{nlp}(a \rightarrow a'))\right).$$

Here a (resp. a') is the original (resp. new) A-value, and $\text{nlp}(a \rightarrow$

a') is the natural language description of the value modification, denoted as $a \to a'$, *e.g.,* "the A-attribute value is modified from a to a'", and P_{LLM} is the probability estimated by LLMs. Intuitively, $\rho_A(a, a')$ assigns a lower perplexity score for more plausible and feasible modification. In this paper, we only consider modification $a \to a'$ whose $\rho_A(a, a')$ is low enough *w.r.t.* a threshold.

Appendix D: Comparison of Rule-based Methods

We compare the expressivity of different rule-based methods in Table 5, by summarizing their patterns/predicates/consequences.

Rules are divided into two categories: rules for graph data and rules for tabular data. For each type of rules, we discuss (1) whether they support graph patterns (*e.g.,* general patterns or restricted patterns)? (2) What types of predicates the rules supports, *e.g.,* constant predicates, variable predicates, inequality predicates ($\neq, \geq, \leq, >, <$), ML predicates, 1WL predicates and negated predicates? (3) What consequences of the rules and the primary purposes of the rules are.

Rules for graph data. As shown in Table 5, among all the prior graph rules, RxGNNs are the most expressive ones, by supporting both general graph patterns and all types of predicates for counterfactual explanation. The most related rules are REPs proposed in Makex [23], but they differ from RxGNNs in the following aspects.

- *Patterns.* RxGNNs are defined on general graph patterns while REPs are defined on restricted star patterns. Note that general patterns are much more expressive than star patterns, albeit at the cost of higher computational complexity, *e.g.,* it only takes PTIME to compute the matches of a given star pattern, but this problem becomes NP-hard for a general pattern.
- *Variable predicates.* RxGNNs support all types of predicates. In contrast, to support efficient validation of REPs, they only partially support variable predicates on the centers and leaves (*i.e.,* vertices without any child vertices) of the star patterns; moreover, each center/leaf can carry at most one variable predicate.
- *Purposes.* Although both RxGNNs and REPs are designed for GNN explanations, REPs are used to generate *factual explanations*, while RxGNNs aim to provide *counterfactual explanations*. The different purposes of REPs and RxGNNs call for different designs and methodologies. The former identifies key substructures or features as sufficient conditions supporting the prediction, without modifying the graph; this does not necessarily require negated predicates. In contrast, the latter requires suggesting appropriate modifications to the input graphs as necessary conditions; we need rules that support negated predicates.

To give concrete illustration about the expressivity for RxGNNs, consider $\varphi_1 = Q_1[\bar{x}, x_0](X_1 \to \neg\mathcal{M}_1(x_0))$ in Example 1, where (1) Q_1 is a general pattern and (2) X_1 involves (a) constant predicate x_1.headcount = 1, (b) variable predicate x_4.GPA > x_0.GPA, (c) comparison predicate x_0.#experience < 3 years, (d) ML predicate $\neg\mathcal{M}_1(x_0)$ and (e) negated 1WL predicate $\neg 1\text{WL}(x_4)$. Note that φ_1 *cannot* be expressed by any existing rules, as elaborated as follows.

REPs [23], TIEs [22] and GCRs [24] sacrifice their expressivity for better efficiency, by supporting a special case of graph patterns and restricted (variable) predicates. Although GARs [30] and TACOs [31] partially overcome these limitations by supporting general graph patterns and more predicates, they still lack an important type of predicates, *i.e.,* 1WL predicates. As argued in [23], GNNs are at most as powerful as the 1-WL test in distinguishing graph structures and moreover, most GNNs are based on 1-WL. Hence 1-WL can explain the behaviors of GNNs in principle. Without supporting 1WL predicates, it is hard to justify that a graph pattern together with simple arithmetic comparisons (*i.e.,* $x.A \oplus y.B$ and $x.A \oplus c$) have enough expressive power to explain complex GNNs. GFDs [19] and GEDs [32] inherit the above issue, by supporting an even more restricted subset of predicates, and worse still, GPARs [34] do not support any predicates and are defined with only graph patterns.

Rules for tabular data. Rules are also ubiquitous in tabular data for *e.g.,* ML explanations/understanding [14, 39, 75], data quality [18, 26, 27, 33] and database design [90]. However, the inherent complexity and structure of graph data make it very challenging for these tabular rules to effectively explain GNNs, which operates on graph-structured data via message passing that involves vertices and edges for representing complex relationships. In particular, although [14, 39, 75] aim to explain/understand ML models (non-GNNs), they only support constant predicates, leading to the same issue of limited expressivity as stated above for GFDs [19] and GEDs [32].

Appendix E: Cover Computation

We start with the following notions for defining a cover.

Cover. We say that a graph G *satisfies* φ, denoted by $G \models \varphi$, if for all matches h of $Q[\bar{x}, x_0]$ in G such that if $h \models X$, then $h \models \varphi$. We write $G \models \Sigma$ for a set Σ of RxGNNs if for all $\varphi \in \Sigma$, $G \models \varphi$.

To remove redundant RxGNNs that are logical consequence of other RxGNNs, we use the implication analysis [28], which has been widely used to decide whether a rule is implied by a set Σ of other rules. Specifically, a set Σ of RxGNNs implies another RxGNN φ, denoted by $\Sigma \models \varphi$, if for all graphs G, $G \models \Sigma$ implies $G \models \varphi$. A *cover* Σ^c of a set Σ of RxGNNs is a subset Σ^c of Σ such that (1) $\Sigma^c \models \Sigma$, *i.e.,* $\Sigma^c \models \varphi$ for all $\varphi \in \Sigma$, and (2) $\Sigma^c \setminus \{\varphi\} \not\models \Sigma^c$ for any $\varphi \in \Sigma^c$.

Problem. The problem of cover computation is stated as follows.

- *Input*: A set Σ of RxGNNs discovered.
- *Output*: A cover set Σ^c of Σ such that (1) $\Sigma^c \models \Sigma$, *i.e.,* $\Sigma^c \models \varphi$ for all $\varphi \in \Sigma$, and (2) $\Sigma^c \setminus \{\varphi\} \not\models \Sigma^c$ for any $\varphi \in \Sigma^c$.

Note that implication analysis (*i.e.,* decide whether a rule φ is implied by a set Σ of rules, denoted as $\Sigma \models \varphi$) is embedded in cover computation. Below we first review the characterization of implication in [28, 29] and adapt it for RxGNNs, which take the same classifier $\mathcal{M}$ in the consequences, followed by the (parallel) algorithm for cover computation along the same line as in [28, 29].

Characterization. For $\varphi = Q[\bar{x}, x_0](X \to \neg\mathcal{M}(x_0))$, the set of RxGNNs embedded in Q, denoted by Σ_Q, consists of RxGNNs of the form $\varphi' = Q'[\bar{x}', x_0](f(X') \to \neg\mathcal{M}(x_0))$ such that (a) there exists an RxGNN $\varphi_m = Q'[\bar{x}', x_0](X' \to \neg\mathcal{M}(x_0))$ in Σ and (b) there exists an homomorphic mapping f from Q' to Q, where $f(X')$ is obtained by replacing x by $f(x)$ for every vertex x appearing in X'.

Intuitively, Σ_Q consists of φ' in Σ s.t. Q' can be mapped to Q and hence, φ' is enforced on every match of Q in a graph satisfying Σ.

For each Σ_Q and a given RxGNN $\varphi = Q[\bar{x}, x_0](X \to \neg\mathcal{M}(x_0))$, we define a set closure (Σ_Q, X) as follow:

(a) all predicates in X are in closure (Σ_Q, X); and

Data	Rules	Patterns	Predicates						Consequences	Purpose of Rules
			Constant?	Variable?	beyond equality (*e.g.*, $\neq, \geq, \leq$)?	ML?	1WL?	ML/1WL Negation?		
Graph	RxGNNs	general	✓	✓	✓	✓	✓	✓	$\neg M(x_0)$	counterfactual explanation
	REPs [23]	star only	✓	partially	✓	✓	✓	×	$M(x_0)$	factual explanation
	TIEs [22]	star only	✓	partially	✓	✓	×	×	(x, likes, y) (*i.e.,* recommend item y to user x)	improve ML recommendation
	GARs [30]	general	✓	✓	✓	✓	×	×	all supported predicate types	association analyses
	GCRs [24]	star only	✓	partially	✓	✓	×	×	$x.A = y.B$ or $x.A = c$	graph data cleaning
	GFDs [35]	general	✓	✓	×	×	×	×	$x.A = y.B$ or $x.A = c$	graph data cleaning
	TACOs [31]	general	✓	✓	✓	✓	×	×	$((x, l, y), \tau)$ (*i.e.,* the event specified by (x, l, y) will take place within time window τ)	association analyses for temporal event prediction
	GPARs [34]	general	×	×	×	×	×	×	(x, l, y) (*i.e.,* the existence of edge (x, l, y))	association analyses
	GEDs [32]	general	✓	✓	×	×	×	×	$x.A = y.B$ or $x.A = c$	graph data cleaning
Tabular	[14, 75]	×	✓	×	✓	×	×	×	$M(t) = i$	understand ML predictions
	RCs [39]	×	✓	×	✓	×	×	×	$\neg M(t)$	counterfactual explanation
	REEs [33]	×	✓	✓	✓	✓	×	×	all supported predicate types	tabular data cleaning
	DCs [18]	×	✓	✓	✓	×	×	×	all supported predicate types	tabular data cleaning
	CFDs [27]	×	✓	✓	×	×	×	×	$t.A = s.B$ or $t.A = c$	conflict resolution
	MDs [26]	×	×	✓	×	✓	×	×	$t.\text{id} = s.\text{id}$	entity resolution
	FDs [90]	×	×	✓	×	×	×	×	$t.A = s.A$	database design

Table 5: Expressivity comparison for different rules

(b) for every $\varphi' = Q'[\bar{x}', x_0](X' \to \neg M(x_0))$ in Σ_Q, $\neg M(x_0)$ is in closure (Σ_Q, X) if either $X' = \emptyset$ or $X' \neq \emptyset$ and all predicates in X' can be deduced from closure (Σ_Q, X) via equality transitivity.

One can verify that given Σ_Q, closure (Σ_Q, X) can be computed in PTIME. We say that closure (Σ_Q, X) is *conflicting* if there exist an attribute $x.A$ and two distinct constants c and d such that both $x.A = c$ and $x.A = d$ can be both deduced from closure (Σ_Q, X).

The implication of RxGNNs is characterized as follows. For any set Σ of RxGNNs and $\varphi = Q[\bar{x}, x_0](X \to \neg M(x_0))$, $\Sigma \models \varphi$ *if and only if* closure(Σ_Q, X) *is conflicting or* $\neg M(x_0)$ *is in* closure(Σ_Q, X). Intuitively, it says that either (a) Σ and (Q, X) are inconsistent and hence, $\neg M(x_0)$ is their "logical consequence"; or (b) closure(Σ_Q, X) is consistent and $\neg M(x_0)$ is enforced by Σ and (Q, X).

Sequential cover computation. By utilizing the above characterization, a sequential cover computation algorithm works by inspecting RxGNNs in Σ one by one. For each $\varphi \in \Sigma$, it checks whether $\Sigma \setminus \{\varphi\} \models \varphi$ based on the characterization; if so, it removes φ from Σ, until no more φ can be removed and the resulting Σ is returned as Σ^c.

Following a similar argument in [28, 29], one can verify that this algorithm correctly compute the cover Σ^c of Σ since (a) for each φ in Σ, it is checked based on the characterization of implication and (b) when the algorithm terminates, no φ can be removed from Σ^c.

Parallel computation. We parallelize the sequential algorithm.

More specifically, we partition the set Σ of RxGNNs into m groups $\Sigma_{Q_1}, \Sigma_{Q_2}, \ldots \Sigma_{Q_m}$, where each Σ_{Q_i} $(i \in [1, m])$ is the set of RxGNNs Σ that pertain to the same pattern Q_i. It suffices to check the implication of RxGNNs within each group, since the implication checking is *pairwise independent* among the groups [28, 29].

For each Σ_{Q_i}, we create a work unit and collect it in a set W. The workload in W are evenly distributed to all workers. Upon receiving the assigned work unit W_j, each worker P_j invokes the sequential algorithm above to compute local cover Σ_j^c in parallel. When all work units are processed, the union of all Σ_j^c's are returned.

The parallel cover computation is also parallely scalable [28, 29].

Appendix F: Additional Details about Datasets

We give the meanings of vertices, edges, attributes for each dataset.

Loan. The Loan dataset contains comprehensive information about loan applications and their associated risk factors, designed for loan default prediction and risk assessment tasks. The dataset captures multi-dimensional relationships between borrowers, their geographical locations, professional backgrounds, and loan characteristics through a heterogeneous graph structure. It enables advanced analysis of loan approval patterns, risk factors, and predictive modeling for financial institutions to make informed lending decisions.

Vertices. Loan has the following types of vertices.

- Loan vertices, which represent individual loan applications or loan products within the system.
- Applicant vertices, which represent loan applicants or borrowers seeking financial assistance.
- City vertices, which represent geographical city locations where applicants reside or loans are processed.
- State vertices, which represent state-level geographical divisions for broader regional analysis.
- Profession vertices, which represent occupational categories or job types of loan applicants.

Edges. Loan has the following types of edges.

- Applicant-Loan. It indicates which applicant has applied for or is associated with a specific loan.
- Applicant-City. It shows the residential location of the applicant.
- City-State. It describes which state each city belongs to.
- Applicant-Profession. It connects applicants to their respective occupational categories.

Attributes. The vertices of Loan carry the following attributes.

- VertexId: A unique identifier assigned to each vertex for individual tracking and reference purposes across all vertex types.
- Label: A categorical indicator used to distinguish vertex types (*i.e.,* Loan, Applicant, City, State or Profession).
- Income: The financial income level of the loan applicant, representing their earning capacity and financial stability.
- Age: The age of the loan applicant.
- Experience: The total work experience of the applicant.
- Married_Single: The marital status of the applicant.
- House_Ownership: The housing ownership status of the applicant (owned, rented, etc.), indicating the asset ownership.
- Car_Ownership: The vehicle ownership status of the applicant, representing additional asset ownership and financial capacity.
- Profession: The specific occupational category of the applicant.
- Current_Job_Year: The number of years the applicant has been

in their current job position, indicating employment stability.
- Current_House_Year: The duration the applicant has been residing at their current address.
- Risk_Flag: A binary ground truth classification label indicating whether the loan application poses a high risk (1) or low risk (0).
- GNNPrediction: The GNN prediction for a specific loan.

Claim. The Claim dataset contains real-world medical insurance information designed for fraud detection in healthcare claims. It encompasses approximately 200,000 beneficiaries, over 5,000 healthcare providers, and around 550,000 medical insurance claims, with 38.1% of claims identified as fraudulent. The dataset features meticulous fraud labels and timestamps annotated by experts, providing typicality and authority for medical fraud detection research.

Vertices. Claim has the following types of vertices.
- Claim vertices, which represent individual medical insurance claims submitted for reimbursement.
- Beneficiary vertices, which represent patients or individuals who receive medical services and file insurance claims.
- Provider vertices, which represent healthcare providers, hospitals, or medical facilities that deliver services.
- DiagnosisGroup vertices, which represent categories of medical diagnoses or disease classifications.

Edges. Claim has the following types of edges.
- Beneficiary-Claim. It indicates which beneficiary has filed a specific medical insurance claim.
- Provider-Claim. It shows which healthcare provider is associated with or has submitted a particular claim.
- Claim-DiagnosisGroup. It connects claims to their corresponding medical diagnosis categories or disease groups.

Attributes. The vertices of Claim carry the following attributes.
- VertexId: A unique identifier assigned to each vertex for individual tracking and reference purposes across all vertex types.
- Label: A categorical indicator used to distinguish vertex types (*i.e.,* Claim, Provider, DiagnosisGroup or Beneficiary).
- InscClaimAmtReimbursed: The reimbursed value for the claim.
- ClmAdmitDiagnosisCode: The medical diagnosis code assigned upon hospital admission.
- DeductibleAmtPaid: The deductible amount paid by the beneficiary, indicating the out-of-pocket expense responsibility.
- Inpatient: A binary indicator specifying whether the medical service was provided as inpatient care or outpatient treatment.
- IsGroupFraud: A binary flag indicating whether the claim is part of a coordinated group fraud scheme involving multiple parties.
- Gender: The gender of the beneficiary.
- Race: The racial demographic information of the beneficiary.
- RenalDiseaseIndicator: A binary indicator showing whether the beneficiary has renal (kidney) disease.
- State: The state information of the beneficiary/claim.
- County: The county information of the beneficiary/claim.
- numDiseases: The total number of diseases or medical conditions affecting the beneficiary.
- IsDead: A binary flag indicating whether the beneficiary is alive.
- IsFraud: A binary ground truth classification label indicating whether the claim is fraudulent (1) or legitimate (0).
- GNNPrediction: The GNN prediction for a specific claim.

Transaction. The Trans dataset contains comprehensive information about financial transactions for fraud detection. It comprises over 6.36 million transaction records, providing a rich and diverse collection of transactional data for analysis and modeling. Each transaction includes detailed information about the parties involved, monetary amounts, account balances, and fraud labels, making it particularly valuable for developing and evaluating fraud detection algorithms using graph-based machine learning approaches.

Vertices. Trans has the following types of vertices.
- Transaction vertices, which represent individual financial transactions within the system.
- Account vertices, which represent user accounts or entities participating in transactions.

Edges. Edges in Trans represent the relationships between Account vertices and Transaction vertices, indicating the flow of funds and transaction participation patterns within the financial network.

Attributes. The vertices of Trans carry the following attributes.
- VertexId: A unique identifier assigned to each vertex (transaction or account) for individual tracking and reference purposes.
- Label: A categorical indicator used to distinguish vertex types (*i.e.,* Transaction or Account).
- Type: The category of financial transaction, including types such as PAYMENT, TRANSFER, CASH_OUT, DEBIT, etc., which help us classify different transaction behaviors.
- Amount: The monetary value involved in the transaction, representing the financial magnitude and scale of each transaction.
- oldbalanceOrig: The account balance of the originating entity before the transaction occurs.
- newbalanceOrig: The updated account balance of the originating entity after the transaction is completed.
- oldbalanceDest: The account balance of the destination entity before receiving the transaction.
- newbalanceDest: The updated account balance of the destination entity after receiving the transaction.
- isFraud: A binary ground truth classification label indicating whether the transaction is fraudulent (1) or legitimate (0).
- GNNPrediction: The GNN prediction for a specific transaction.

Amazon. The Amazon dataset describes an Amazon clothing e-commerce network capturing the interactions between users, products, and reviews within the online apparel marketplace. It encompasses commercial relationships, user fashion preferences, and clothing product information, enabling analysis of purchasing patterns, review authenticity, and product recommendation.

Vertices. Amazon has the following types of vertices.
- Product vertices, which represent individual clothing products available for purchase on the Amazon platform.
- Review vertices, which represent individual customer reviews and ratings submitted for products.
- User vertices, which represent customers who purchase products or submit reviews on the platform.

Edges. Amazon has the following types of edges.

- User-Review. It gives the ownership between users and reviews.
- User-(buy)-Product. It indicates that a user has bought a product.
- User-(comment)-Product. It indicates that the user has commented on a product but not purchased yet.
- Product-Review. It connects products to the reviews they have received from customers.

Attributes. The vertices of Amazon carry the following attributes.

- VertexId: A unique identifier assigned to each vertex for individual tracking and reference purposes across all vertex types.
- Label: A categorical indicator used to distinguish between the three different vertex types (*i.e.,* Product, Review or User).
- reviewText: The detailed content of the customer review.
- reviewSummary: A concise summary or title of the review.
- reviewScore: A numerical rating score given by the reviewer, typically on a scale representing satisfaction level.
- productTitle: The title of the product as listed on the platform.
- brand: The manufacturer or brand name of the product.
- price: The selling price of the product.
- reviewTime: The timestamp when the review was submitted.
- reviewerName: The identifier of the customer who submitted the review.
- style: The specific attributes of the clothing item such as color, size, material, fit, or design options.
- vote: The number of votes/likes received by a review, indicating its perceived usefulness by other users.
- description: The product description provided by the seller.
- category: A multi-class ground truth classification label indicating product categories, *i.e.,* men's clothing (0), women's clothing (1), women's underwear (2), outdoor sports (3), jewelry (4).
- GNNPrediction: The GNN prediction for a specific product.

Appendix G: Details about Baselines

This section presents more details about the baselines. Note that not all baselines are designed for vertex classification on heterogeneous graphs where vertices can carry both labels and attributes. For example, in our problem, even two vertices are both labeled as user vertices, they can still carry different attributes, *e.g.,* ages, names and genders; this may not be the case in other settings, *e.g.,* graph classification for molecules, where each vertices are atoms.

Thus we will adapt the baselines for (a) vertex classification and (b) attributed graphs. Note that to tackle labels and attributes on vertices, existing GNN techniques typically associate vertices with feature vectors (in the form of numerical encodings, *e.g.,* word embedding or one-hot encoding) [16]. We follow this in our adaptations.

Local explanation. These methods are designed to generate graph perturbations that flip GNN predictions. CF-Exp operates exclusively through edge removal, while InduCE and CFGraph support both edge removal and edge insertion. Clear employs a VAE framework to encode the input graph into a latent representation, subsequently utilizing a decoder to reconstruct a counterfactual graph that maintains structural fidelity to the input graph while achieving the target prediction flip. GVEX is originally designed for graph classification tasks. To adapt it for vertex classification tasks, it requires constructing a receptive field graph for each training vertex. After that, for each input graph, GVEX computes Jacobian matrices to identify vertices with maximum feature influence, and conducts gradual exploration through graph bisection that partitions the input graph into two subgraphs with opposing prediction outcomes.

Since most baselines do not consider modifications on vertex features, we implemented a vertex-level masking scheme [96] that operates under the same budgets as edge modifications for fair comparison, *i.e.,* we allow the baselines to apply a feature mask on the feature vectors of some vertices to perturb the model prediction.

However, we argue that feature masking is typically not as intuitive as the dependencies on raw attributes in RxGNNs, where individual attributes can be clearly highlighted in the explanation. As commented in [23], applying the same feature mask to multiple vertices indicates that "the features on different vertices are of equal importance in its explanations, which is, however, often not true".

To optimize the performance of all counterfactual methods, we implemented the following strategy. We apply graph perturbations one at a time (whenever feasible). If the current perturbations applied already successfully swap the predictions, we stop directly. This procedure continues until no more perturbation can be applied.

In particular, we would like to express our sincere gratitude to the authors of GVEX for their extensive discussions when we adapted their methods. Since GVEX is particularly designed for graph classification, it incurs performance degradation when adapting to vertex classification where the receptive field graph constructed for each training vertex (*i.e.,* the pivot vertex) is much larger than their original settings in [16]. This is mainly because that GVEX requires identifying influential subgraphs, which incurs a much higher computational complexity than identifying edges to insert/remove like most other methods. Moreover, since GVEX requires computing the Jacobian matrix to measure feature influence, it has to compute the pairwise "influence" of all vertices in each receptive field graph (*i.e.,* $O(n^2)$ where n is the number of vertices in the graph constructed for a training vertex). Although we can approximate this by only considering the "influence" of all vertices to the pivot (*i.e.,* reduce from $O(n^2)$ to $O(n)$), GVEX is still slower than NEX by around two orders of magnitudes. In our experiments, we mainly adopted the original setting in [16] and followed the suggestions of the authors.

Global explanation. Both XGNN and GNNInt generate discriminative graph patterns to characterize model behavior: XGNN employs reinforcement learning to maximize specific predictions, while GNNInt uses a probabilistic generative approach with numerical optimization to learn explanation graph distributions. Both methods were originally developed for molecular graph analysis and focus exclusively on topological pattern generation without incorporating vertex features. GVEX summarizes global graph patterns from local explanation subgraphs. CFGraph generates global explanations by aggregation of multiple counterfactual instances, by analyzing the structural differences and edge modification patterns across instances to identify discriminative structures.

Since XGNN and GNNInt only produce topological patterns devoid of vertex feature representations, we learn specific embeddings for each type of pattern vertices. For GVEX and CFGraph, we extract vertex embeddings from the input graph and apply clustering to get representative embeddings for each type of pattern vertex.