EXPLAINABLE GNN-BASED MODELS OVER KNOWLEDGE GRAPHS

Anonymous authors
Paper under double-blind review

ABSTRACT

Graph Neural Networks (GNNs) are often used to realise learnable transformations of graph data. While effective in practice, GNNs make predictions via numeric manipulations in an embedding space, so their output cannot be easily explained symbolically. In this paper, we propose a new family of GNN-based transformations of graph data that can be trained effectively, but where all predictions can be explained symbolically as logical inferences in Datalog—a well-known knowledge representation formalism. Specifically, we show how to encode an input knowledge graph into a graph with numeric feature vectors, process this graph using a GNN, and decode the result into an output knowledge graph. We use a new class of *monotonic* GNNs (MGNNs) to ensure that this process is equivalent to a round of application of a set of Datalog rules. We also show that, given an arbitrary MGNN, we can extract automatically a set of rules that completely characterises the transformation. We evaluate our approach by applying it to classification tasks in knowledge graph completion.

1 INTRODUCTION

Knowledge graphs (KGs) (Hogan et al., 2021) are graph-structured knowledge bases that describe an application domain as a collection of entities and their relationships. They can be captured in formats such as the Resource Description Framework (RDF) or labelled property graphs, and they are extensively used in applications such as Web search, recommender systems, social networks, fraud detection, and data integration. Many of these applications can be conceptually understood as realising a knowledge graph transformation. For example, in a recommender system, we can represent interactions between users and items in graph format (Aggarwal, 2016) and further integrate this data with external KGs to improve accuracy, diversity, and interpretability of recommendations (Zhang et al., 2016; Wang et al., 2018; 2019a), and we can see the system as realising a transformation of the input graph into another graph containing recommended user-item interactions. Moreover, a knowledge graph completion system (Rossi et al., 2021; Bordes et al., 2013; Teru et al., 2020) can be understood as transforming an input graph into another graph containing relationships missing in the input. Usually, the transformation is not given explicitly, but is learned from labelled examples.

Graph Neural Networks (GNNs) (Scarselli et al., 2009; Liu & Zhou, 2020) are one of the most popular ML models for graph-structured data. They are usually applied to KGs in three stages (Hamilton et al., 2017). First, the input knowledge graph is *encoded* into an *embedding space* as another graph whose vertices are labelled with numeric feature vectors. Second, the encoded graph is processed by a GNN consisting of several layers. For each layer, the GNN updates the feature vectors based on the learned model parameters and the feature vectors of the previous layer. Finally, the feature vectors of the output layer are *decoded* to the output knowledge graph.

While such approaches can be very effective in practice, GNNs compute their predictions through complex numeric calculations in the embedding space, so the transformation can be difficult to understand by humans. For example, if a system recommends novel *The Idiot* to user Alice, it would be useful to know that this was made because Alice liked *Crime and Punishment*, and both novels were written by Fyodor Dostoevsky. There is a growing interest within the ML community in explaining the predictions of a GNN. Many of the existing methods have focused on identifying parts of the graph in the embedding space that are most relevant to a given prediction (Ying et al., 2019; Luo et al., 2020; Lin et al., 2021). Such methods, however, do not ex-

plain predictions *symbolically*—that is, by showing how each prediction can be derived from the input knowledge graph using logical rules of inference in a knowledge representation formalism. Datalog (Abiteboul et al., 1995; Ceri et al., 1989) is a prominent rule-based formalism where a domain of interest is described using 'if–then' rules. When applied to a set of facts, the rules derive new facts. For example, we can represent background knowledge about literature using facts $author(Crime_and_Punishment, Dostoevsky)$ and $author(The_Idiot, Dostoevsky)$, and we can represent past user-item interactions using fact $likes(Alice, Crime_and_Punishment)$. Then, rule $likes(x, y_1) \wedge author(z, y_1) \wedge author(z, y_2) \rightarrow recommend(x, y_2)$ provides a symbolic explanation for the fact $recommend(Alice, The_Idiot)$ since applying the rule to the input facts derives the target output fact. Such explanations foster trust in a system's predictions, ensure compliance with norms, and allow one to verify standards of fairness. Thus, developing methods for providing such symbolic explanations is an important and challenging research problem.

The main contribution of this paper is a new family of GNN-based transformations of knowledge graphs that can be trained from examples as usual, but whose predictions can be explained symbolically using Datalog rules. Our transformation consists of the three steps mentioned earlier: encoding an input KG into an embedding space, applying a GNN, and decoding the result back into an output KG. These steps, however, were carefully crafted to ensure that the transformation can always be described symbolically. In particular, we use a novel class of *monotonic GNNs* (MGNNs), which we designed to ensure an important property: when values in input feature vectors increase, no value of an output feature vector decreases. Intuitively, this ensures that our transformation 'behaves like Datalog'. To capture this intuition formally, we show that our transformation is *monotonic under homomorphisms*—a well-known property that captures key aspects of Datalog rule application.

In contrast to the related approaches (Teru et al., 2020; Hamaguchi et al., 2017; Schlichtkrull et al., 2018; Wang et al., 2019b), all predictions of our transformation can be explained using Datalog rules. Specifically, for an arbitrary MGNN, we can produce an *equivalent* set of rules so that, on any knowledge graph, the rules and the MGNN-based transformation produce the same facts. This opens the door to new ways of using MGNNs: when developing Datalog rules manually is difficult, we can train an MGNN model in a supervised manner and use it to provide further predictions, but all such predictions can be explained using rules. For example, we could train an MGNN on a graph containing examples of credit applications and then decide new applications by applying the model to a different graph; this is analogous to how deep learning is already used in practice. However, unlike the existing approaches, we can explicate the rules that are implicitly captured by the model and use them to *fully* explain each credit decision. Thus, we integrate subsymbolic and symbolic AI in a single framework, allowing applications to use the best of both worlds.

We demonstrate the effectiveness of our approach in practice by applying it to classification tasks in knowledge graph completion. Specifically, using several well-known benchmarks, we compare the performance of our system with that of DRUM (Sadeghian et al., 2019) and AnyBURL (Meilicke et al., 2019), both state-of-the-art knowledge graph completion systems able to also generate logical rules. We show that, in addition to providing an exact correspondence between the model and the extracted logical rules, our technique also offers very competitive performance.

Proofs of all our formal statements are given in the appendix, and the source code of our implementation is available as supplementary material.

2 An MGNN-Based Transformation of Knowledge Graphs

We now describe our GNN-based transformation of knowledge graphs. In this paper, we see a KG as a *dataset*, which is a finite set of facts of the form $A(a)$ or $R(a_1, a_2)$, where A is a *unary predicate*, R is a *binary predicate*, and a, a_1, and a_2 are *constants*. Constants represent entities from an application domain, unary predicates typically represent entity types, and binary predicates represent relationships between entities. KGs in a format such as RDF can be transformed into such a dataset in the obvious way. For example, the statement that *The Idiot* is a novel that was written by Fyodor Dostoevsky can be represented as facts $novel(The_Idiot)$ and $author(The_Idiot, Dostoevsky)$. For D a dataset, $\mathsf{cn}(D)$ is the set of constants occurring in D.

Our transformation is realised by an operator $T_{\mathcal{M}}$ on datasets, which is parameterised by a GNN $\mathcal{M}$. A dataset D is first encoded into a graph G_D in the embedding space, where vertices are labelled

with numeric feature vectors; then, G_D is processed by $\mathcal{M}$; finally the result is decoded to the output dataset $T_{\mathcal{M}}(D)$. In Section 3, we show that, by restricting $\mathcal{M}$ to be a GNN of a special form, called *monotonic*, we can ensure that $T_{\mathcal{M}}$ is equivalent to a round of Datalog rule application; this, in turn, allows us to automatically extract an equivalent set of rules from $T_{\mathcal{M}}$. The embedding space consists of *coloured graphs*, which extend the standard notion of undirected graphs in two ways. First, each vertex is labelled by a numeric feature vector of some fixed dimension; this is necessary in order to allow the coloured graph to be processed by $\mathcal{M}$. Second, each edge in the graph is assigned a colour, which allows us to distinguish different types of connections in the graph.

To illustrate our ideas, we describe the encoding of dataset $D = \{A(a), R(a, b), B(d)\}$. First, we introduce vertices v_a, v_b, and v_d in G_D to represent constants a, b, and d. Intuitively, vertex v_a will represent all facts of the form $X(a)$ for some unary predicate X, and analogously for the other vertices. Second, we introduce vertices $v_{a,b}$ and $v_{b,a}$; intuitively, these will represent all facts of the form $Y(a, b)$ and $Y(b, a)$, respectively, for some binary predicate Y. Note that, since constants a and d do not occur together in a binary fact in D, the encoding of D does not contain vertices $v_{a,d}$ or $v_{d,a}$. Third, we order all predicates arbitrarily and assign to each predicate a position in the feature vectors labelling the vertices of G_D. In our example, let us order the predicates as A, B, R; then, we translate $A(a)$, $B(d)$, and $R(a, b)$ by respectively setting to one the first, second, and third positions of the feature vectors labelling v_a, v_d, and $v_{a,b}$, respectively. Fourth, we connect all vertices of G_D that refer to common constants by edges of appropriate colour. For example, v_a and $v_{a,b}$ both refer to constant a, so we connect them by an edge of colour c_1, thus indicating that a occurs first in the constant pair of $v_{a,b}$. Analogously, we connect $v_{a,b}$ and v_b by an edge of colour c_2. Thus, colours allow us to distinguish different types of joins between facts in D. Vertices with no constants in common are not connected; for example, v_d is not connected to any vertex (including itself).

To formalise these intuitions, we fix $\mathsf{Col} = \{c_1, c_2, c_3, c_4\}$ as the set of four *colours*. Also, we assume that each dataset and all derived facts will use only finitely many predicates. Thus, we fix a list $A_1, \ldots, A_\epsilon$, $R_{\epsilon+1}, \ldots, R_\delta$ of all available unary and binary predicates; here, ϵ is the number of unary predicates, and δ is the total number of predicates. In what follows, we consider vectors and matrices over the set $\mathbb{R}$ of reals and its nonnegative subset $\mathbb{R}^+$. For $\mathbf{a}$ a vector, $(\mathbf{a})_i$ is the i-th element of $\mathbf{a}$. We apply functions to vectors and matrices element-wise; for example, for $\mathbf{a}_1, \ldots, \mathbf{a}_n$ vectors of equal dimension, $\max\{\mathbf{a}_1, \ldots, \mathbf{a}_n\}$ is the vector whose i-th element is $\max\{(\mathbf{a}_1)_i, \ldots, (\mathbf{a}_n)_i\}$. Finally, a function $\sigma : \mathbb{R} \to \mathbb{R}$ is *monotonically increasing* if $x < y$ implies $\sigma(x) \leq \sigma(y)$, and function σ is *Boolean* if it ranges over $\{0, 1\}$.

The following definition describes the structure of our embedding space.

Definition 1. *A* (Col, δ)-graph *is a tuple* $\langle V, \{E_c\}_{c \in \mathsf{Col}}, \lambda\rangle$*, where* V *is a finite set of vertices; for each* $c \in \mathsf{Col}$*,* E_c *is a set of* c*-coloured undirected edges (possibly including self-loops) over* V*; and* labelling λ *assigns to each* $v \in V$ *a vector* $\lambda(v)$ *of dimension* δ*. If* λ *is implicitly understood,* $\lambda(v)$ *is written as* $\mathbf{v}$*. For* $c \in \mathsf{Col}$ *and* $v \in V$*,* $N_c(v)$ *is the set of vertices connected to* v *by an edge in* E_c*.*

We are now ready to define our encoding of an arbitrary dataset D into a (Col, δ)-graph G_D.

Definition 2. *The* encoding *of a dataset* D *is the* (Col, δ)*-graph* G_D *that contains a vertex* v_a *for each constant* $a \in \mathsf{cn}(D)$*, and vertices* $v_{a,b}$ *and* $v_{b,a}$ *for all (not necessarily distinct) constants* a *and* b *that occur together in a binary fact in* D*. For each vertex* $v_{a,b}$*,* G_D *contains an edge of colour* c_1 *connecting* $v_{a,b}$ *and* v_a*, an edge of colour* c_2 *connecting* $v_{a,b}$ *and* v_b*, an edge of colour* c_3 *connecting* $v_{a,b}$ *and* $v_{b,a}$*, and an edge of colour* c_4 *connecting* v_a *and* v_b*. For each* v_a*, labelling* $(\mathbf{v}_a)_i$ *is* 1 *if* $A_i(a) \in D$ *and* 0 *otherwise. For each* $v_{a,b}$*, labelling* $(\mathbf{v}_{a,b})_i$ *is* 1 *if* $R_i(a, b) \in D$ *and* 0 *otherwise.*

We process G_D by a graph neural network. As we explain in more detail Section 3, if we are to ensure that our transformation 'mimics' Datalog rule application, we cannot use an arbitrary GNN. Instead, we introduce *Monotonic Graph Neural Networks (MGNNs)*, which restrict GNNs in a way that allows us to attain our objective. Definition 3 also specifies how to apply an MGNN to a coloured graph.

Definition 3. *A* (Col, δ)-monotonic graph neural network *(*MGNN*)* $\mathcal{M}$ *with* $L \geq 1$ layers *is a tuple* $\langle \{\mathbf{A}^\ell\}_{1 \leq \ell \leq L}, \{\mathbf{B}^\ell_c\}_{c \in \mathsf{Col}, 1 \leq \ell \leq L}, \{\mathbf{b}^\ell\}_{1 \leq \ell \leq L}, \sigma, \mathsf{cls}\rangle$*. For each* $\ell \in \{1, \ldots, L\}$ *and each* $c \in \mathsf{Col}$*,* $\mathbf{A}^\ell$ *and* $\mathbf{B}^\ell_c$ *are matrices over* $\mathbb{R}^+$ *of dimension* $m^\ell \times n^\ell$*, and* $\mathbf{b}^\ell$ *is a vector over* $\mathbb{R}$ *of dimension* n^ℓ *such that* $n^1 = m^L = \delta$*, and* $m^{\ell-1} = n^\ell$ *for each* $\ell \in \{2, \ldots, L\}$*. Moreover,* $\sigma : \mathbb{R} \to \mathbb{R}^+$ *is a monotonically increasing* activation *function, and* $\mathsf{cls} : \mathbb{R}^+ \to \{0, 1\}$ *is a monotonically increasing* classification *function. The* dimension *of* $\mathcal{M}$ *is* $\max\{n^1, \ldots, n^L\}$*.*

Applying $\mathcal{M}$ to a (Col, δ)-graph $G = \langle V, \{E_c\}_{c \in \mathsf{Col}}, \lambda \rangle$ induces the sequence $\lambda^0, \ldots, \lambda^L$ of vertex labellings where $\lambda^0 = \lambda$, and for each $\ell \in \{1, \ldots, L\}$ and $v \in V$, the value of $\lambda^\ell(v)$ is given by

$$\mathbf{v}^\ell = \sigma\big(\mathbf{A}^\ell \mathbf{v}^{\ell-1} + \sum_{c \in \mathsf{Col}} \mathbf{B}_c^\ell \max\{\mathbf{w}^{\ell-1} \mid w \in N_c(v)\} + \mathbf{b}^\ell\big). \tag{1}$$

The result $\mathcal{M}(G)$ of applying $\mathcal{M}$ to G is the (Col, δ)-graph with the same vertices and edges as G, but where each vertex $v \in V$ is labelled by $\mathsf{cls}(\lambda^L(v))$.

Intuitively, an MGNN consists of L layers, where each layer $\ell \in \{1, \ldots, L\}$ is defined by a matrix $\mathbf{A}^\ell$, a matrix $\mathbf{B}_c^\ell$ for each colour c, and a bias vector $\mathbf{b}^\ell$; these constitute the learnable parameters of the model. In addition, an MGNN also uses functions σ and cls which are fixed and given explicitly. An MGNN transforms an arbitrary (Col, δ)-graph G into another (Col, δ)-graph $\mathcal{M}(G)$ as follows. Each vertex v in G and each $\ell \in \{0, \ldots, L\}$ is associated with a labelling $\mathbf{v}^\ell$. Labelling $\mathbf{v}^0$ is the labelling $\mathbf{v}$ of v in G, and, for each layer ℓ, vertex v is labelled by a feature vector $\mathbf{v}^\ell$ computed as specified in equation (1). The feature vector of v in the resulting graph is computed by applying the classification function cls to the final labelling $\mathbf{v}^L$. Thus, MGNNs instantiate the GNN framework by (i) taking the maximum of the features of a vertex' neighbours (instead of the more commonly used sum or average), (ii) using nonnegative weights in matrices $\mathbf{A}^\ell$ and $\mathbf{B}_c^\ell$ (but the elements of bias vectors $\mathbf{b}^\ell$ can be negative), and (iii) requiring the activation and classification functions to be monotonically increasing. We discuss the reasons for these restrictions in Section 3.

The result $\mathcal{M}(G_D)$ of applying an MGNN to the encoding G_D of a dataset D can be decoded into the output dataset of the transformation by essentially inverting the encoder. Thus, for each MGNN $\mathcal{M}$, we obtain a distinct operator $T_\mathcal{M}$ that is specified as follows.

Definition 4. *For $\mathcal{M}$ a (Col, δ)-MGNN and D a dataset, $T_\mathcal{M}(D)$ is the dataset containing the fact $A_i(a)$ for each $i \in \{1, \ldots, \epsilon\}$ and each vertex v_a such that $(\mathbf{v}_a)_i = 1$ in $\mathcal{M}(G_D)$, and the fact $R_i(a, b)$ for each $i \in \{\epsilon + 1, \ldots, \delta\}$ and each vertex $v_{a,b}$ such that $(\mathbf{v}_{a,b})_i = 1$ in $\mathcal{M}(G_D)$.*

Note that $T_\mathcal{M}$ can be applied to any dataset D as long as all predicates in D are associated with one of the δ positions in feature vectors—that is, there is no restriction on the number of constants or facts in D. In other words, the dimensions of the matrices in $\mathcal{M}$ do not predetermine the kind of dataset to which $T_\mathcal{M}$ can be applied; in fact, $\mathcal{M}$ can be trained on some dataset and then applied to a different dataset over the same predicates. This makes our transformation ideally suited for *inductive settings* in applications such as KG completion and recommendation (Teru et al., 2020; Hamaguchi et al., 2017; van den Berg et al., 2017), where the datasets to which a trained model is applied may use constants (i.e., *entities*) not mentioned in the datasets on which the model was trained.

3 Extracting An Equivalent Set of Datalog Rules

We now show that, for each MGNN $\mathcal{M}$, we can automatically generate a set of Datalog rules $\mathcal{P}_\mathcal{M}$ such that $T_\mathcal{M}$ and $\mathcal{P}_\mathcal{M}$ derive the same facts on any dataset D. To this end, we first recapitulate the basic definitions of Datalog and identify a property that characterises Datalog reasoning.

A (Datalog) *atom* is an expression of the form $A(t_1)$ or $R(t_1, t_2)$, where A and R are unary and binary predicates, respectively, and each t_i is a constant or *variable*. A (Datalog) *rule* is an implication of the form (2), where H is the atom called the rule's *head*, all B_i are atoms that together form the rule's *body*, and each variable in H occurs in some B_i.

$$H \leftarrow B_1 \wedge \cdots \wedge B_m \tag{2}$$

A (Datalog) *program* is a finite set of rules. A *substitution* ν is a mapping of finitely many variables to constants; for α an atom, $\alpha\nu$ is the result of replacing in α each variable x on which ν is defined with $\nu(x)$. Each rule r of form (2) defines the *immediate consequence* operator T_r on datasets: for D a dataset, $T_r(D)$ contains fact $H\nu$ for each substitution ν satisfying $\{B_1\nu, \ldots, B_m\nu\} \subseteq D$. For $\mathcal{P}$ a program, operator $T_\mathcal{P}$ on datasets is defined by $T_\mathcal{P}(D) = \bigcup_{r \in \mathcal{P}} T_r(D)$.

Our objective is to show that, for each MGNN $\mathcal{M}$, operator $T_\mathcal{M}$ 'behaves analogously' to Datalog rule application. Definition 5 specifies formally what this means in our setting.

Definition 5. *A* homomorphism *from a dataset D to a dataset D' is a mapping h of constants to constants that is defined at least on $\mathsf{cn}(D)$ and that satisfies $h(D) \subseteq D'$, where $h(D)$ is the dataset*

obtained by replacing each constant a in D with $h(a)$ and removing all duplicate facts. An operator T from datasets to datasets is monotonic under homomorphisms *if, for all datasets D and D', each homomorphism from D to D' is also a homomorphism from $T(D)$ to $T(D')$.*

For any constant-free program $\mathcal{P}$, operator $T_\mathcal{P}$ is monotonic under homomorphisms. For example, let $D = \{A(a), R(a, b)\}$ and $\mathcal{P} = \{S(x, y) \leftarrow A(x) \wedge R(x, y)\}$, so $T_\mathcal{P}(D) = \{S(a, b)\}$. First, operator $T_\mathcal{P}$ is *monotonic*: adding facts to D cannot invalidate any consequence of $T_\mathcal{P}$. Indeed, we can add facts to D, but the body of r still matches and the rule derives $S(a, b)$. Second, operator $T_\mathcal{P}$ is resilient to replacement of constants with other constants. For example, let $D' = \{A(c), R(c, c)\}$ be obtained from D by replacing both a and b with c. Rule r 'cannot tell' that the constants have been replaced: we derive $T_\mathcal{P}(D') = \{S(c, c)\}$, which is obtained from $T_\mathcal{P}(D)$ by the same replacements. Monotonicity under homomorphisms combines both of these properties. This notion can be extended to programs with constants, but constant-free rules are sufficient for this paper. We next show that rules can be automatically extracted from any operator that satisfies this property.

Definition 6. *An operator T on datasets* captures *a rule or program α if $T_\alpha(D) \subseteq T(D)$ for each dataset D.*

Proposition 7. *Let T be an operator on datasets, let r be a constant-free rule of form* (2)*, let μ be a substitution that maps each variable of r to a distinct constant, and let $D_r = \{B_1\mu, \ldots, B_m\mu\}$. If T is monotonic under homomorphisms and $H\mu \in T(D_r)$, then T captures r.*

Intuitively, if some operator T captures a rule r, then T_r provides us with a *lower bound* for the consequences of T on an arbitrary dataset—that is, on any dataset, T derives at least as many facts as r (but T can derive more facts). Moreover, if T is monotonic under homomorphisms, Proposition 7 provides us with an effective way of checking whether T captures r: replace each variable in r with a distinct constant, collect the resulting body atoms in a dataset D_r, apply T to D_r, and finally check whether the result contains the replaced head atom of r. Clearly, T captures a constant-free program $\mathcal{P}$ if T captures each rule $r \in \mathcal{P}$, so we can just apply this check to each $r \in \mathcal{P}$ to verify whether T captures $\mathcal{P}$. This provides us with the necessary machinery for rule extraction.

We next show that operator $T_\mathcal{M}$ is monotonic under homomorphisms. This shows that $T_\mathcal{M}$ indeed 'behaves like' Datalog reasoning, and it will allow us to use Proposition 7 for rule extraction.

Theorem 8. *For each (Col, δ)-MGNN $\mathcal{M}$, mapping $T_\mathcal{M}$ is monotonic under homomorphisms.*

Theorem 8 is a direct consequence of the three restrictions on MGNNs mentioned just after Definition 3. For monotonicity, note that adding facts to a dataset D has the effect of possibly adding new vertices and edges and then flipping from 0 to 1 some positions in feature vectors labelling the vertices of G_D; moreover, the elements of $\mathbf{A}^\ell$ and $\mathbf{B}_c^\ell$ are nonnegative so the arguments of σ in equation (1) can only increase; finally, functions σ and cls are monotonically increasing so the elements of the feature vectors in $\mathcal{M}(G_D)$ can only flip from 0 to 1, which is equivalent to deriving more facts in $T_\mathcal{M}(D)$. For the resilience of constant merging, note that replacing, say, constant b with constant a in D is tantamount to merging vertices v_b into v_a in G_D; this will cause vertex v_a to have fewer neighbours, but since we aggregate neighbours' feature vectors by taking the maximum, this cannot change the result of equation (1). Note that, if we aggregated neighbours using a sum (as is common in the related literature), then elements of the feature vectors in $\mathcal{M}(G_D)$ could decrease due to merging, which would invalidate Theorem 8.

Theorem 8 tells us that we can check whether $T_\mathcal{M}$ captures some rule r effectively using Proposition 7. This is a key step of our rule extraction algorithm. In particular, we next show that $T_\mathcal{M}$ can capture only rules of a certain structure, and that the number of such rules is bounded and determined by the dimension of $\mathcal{M}$, the number of layers of $\mathcal{M}$, and the number of predicates δ. Thus, to obtain a program $\mathcal{P}_\mathcal{M}$ deriving the same consequences as $T_\mathcal{M}$, we can enumerate all such rules, apply Proposition 7 to each rule, and keep only the rules that pass this check.

Definition 9 describes the syntactic structure of the rules that $T_\mathcal{M}$ can capture. To this end, we use the well-known notion of a *primal graph* of a conjunction C of atoms, which is the undirected graph (possibly with self-loops) whose vertices are the variables of C, and where an edge connects each pair of (not necessarily distinct) variables x and y such that $R(x, y)$ is in C for some R. Note that the primal graph of an empty conjunction is also empty. A primal graph is a (possibly empty) *tree* if it is connected and acyclic (and, in particular, without self-loops).

Definition 9. *Let c and d be nonnegative integers. A constant-free conjunction C is* (c, d)-tree-like for a variable x *if the primal graph of C is a tree that is either empty or can be rooted at x (i.e., x occurs in C) and, for each vertex v in the tree, the distance i of v from x is at most d and moreover v has at most $c(d - i)$ children.*

A (c, d)-tree-like rule *is a rule of form $A(x) \leftarrow C_x$ or $R(x, y) \leftarrow C_x \wedge B_{x,y} \wedge C_y$, where C_x and C_y are (c, d)-tree-like conjunctions for x and y, respectively, with no variables in common, and $B_{x,y}$ is a nonempty constant-free conjunction of binary atoms of the form $S(x, y)$ or $S(y, x)$.*[1]

For $\mathcal{M}$ a (Col, δ)-MGNN with L layers and dimension k, $\mathcal{P}_\mathcal{M}$ is the program that contains, up to variable renaming, each $(3k, L)$-tree-like rule that is captured by $T_\mathcal{M}$.

We clarify Definition 9 using two examples. First, for $A(x) \wedge R(x, y) \wedge S(x, z)$, the primal graph is a tree where x is the root with two children y and z. Thus, this conjunction is not (1,1)-tree-like: x is at distance zero from itself, so it would be allowed to have at most $1(1 - 0)$ children; however, the conjunction is (2,1)-tree like. Second, the primal graph of $A(x) \wedge R(x, y) \wedge S(x, z) \wedge T(y, z)$ a triangle, so this conjunction is not (c, d)-tree-like for any c and d.

Theorem 10. *For each (Col, δ)-MGNN $\mathcal{M}$ and each dataset D, we have $T_\mathcal{M}(D) = T_{\mathcal{P}_\mathcal{M}}(D)$.*

Apart from allowing us to automatically extract the rules that explain the predictions of an MGNN $\mathcal{M}$, Theorem 10 also describes the expressiveness (i.e., the kind of possible predictions) of $T_\mathcal{M}$. For example, no MGNN can capture rules such as $r = R(x, z) \leftarrow R(x, y) \wedge R(y, z)$. To see this, let $D = \{R(a, b), R(b, c)\}$. For an arbitrary MGNN $\mathcal{M}$, Theorem 10 ensures that all predictions of $T_\mathcal{M}$ involving binary facts are made by rules of the form $R(x, y) \leftarrow C_x \wedge B_{x,y} \wedge C_y$; however, no such rule can derive $R(a, c)$ on D, so we necessarily have $R(a, c) \notin T_\mathcal{M}(D)$ as well.

4 Evaluation

We have evaluated our approach on knowledge graph completion tasks, which involve extending an incomplete knowledge graph represented as a dataset D to its complete version D' by adding missing facts. This problem can be seen as an instance of classification, where the aim is to learn a Boolean function $f(\cdot, \cdot)$ that takes a dataset D and a fact α over a fixed set of unary and binary predicates such that $f(D, \alpha)$ returns *true* if and only if $\alpha \in D'$. We consider the more general *inductive* variant of this task, where the testing dataset may contain constants not occurring in the training or validation datasets (but all predicates in the testing dataset are known before training). Inductive KG completion is more general and challenging than the *transductive* variant (Bordes et al., 2013), where all constants in the testing dataset also occur in the training dataset.

Baselines. We compared our approach with DRUM (Sadeghian et al., 2019) and AnyBURL (Meilicke et al., 2019), two state-of-the-art approaches for knowledge graph completion.

DRUM learns an end-to-end differentiable model that, given a dataset D and a fact α, returns a score representing the likelihood of α belonging to the completion of D; predictions are then computed by applying a threshold to the score. The system also produces a set of *chain* Datalog rules of the form $H(x_1, x_n) \leftarrow B_1(x_1, x_2) \wedge B_2(x_2, x_3) \wedge \cdots \wedge B_{n-1}(x_{n-1}, x_n)$, each associated with a confidence score. These extracted rules approximate the model's predictions, but, unlike in our work, the exact relationship between the predictions and the result of applying the rules is unspecified.

AnyBURL uses path sampling techniques to generate chain rules of a slightly more general form (variables can be replaced by constants). Each rule is associated with a score. Given dataset D and fact α, AnyBURL makes its prediction by applying a threshold to a score computed by first applying the rules on D once and then combining the probabilities associated to all rules deriving α.

Benchmarks and performance metrics. We used the 12 KG completion benchmarks by Teru et al. (2020), which are based on FB15K-237 (Bordes et al., 2013), NELL-995 (Xiong et al., 2017), and WN18RR (Dettmers et al., 2018). Each benchmark provides disjoint datasets $\mathcal{T}$, $\mathcal{V}$, and $\mathcal{S}$ for training, validation, and testing, respectively; statistics are shown in Table 1.

We evaluated the performance of our systems as follows. Each benchmark provides a method for randomly splitting its testing dataset $\mathcal{S}$ into disjoint subsets $\mathcal{S}_I$ and $\mathcal{S}_C$; intuitively, $\mathcal{S}_I$ is the in-

[1] Note that C_x cannot be empty in a rule of form $A(x) \leftarrow C_x$ since x is required to occur in the body.

	FB15K-237				NELL-995				WN18RR			
	v1	v2	v3	v4	v1	v2	v3	v4	v1	v2	v3	v4
Train.	4,245	9,739	17,986	27,203	4,687	8,219	16,393	7,546	5,410	15,262	25,901	7,940
Valid.	489	1,166	2,194	3,352	414	922	1,851	876	630	1,838	3,097	934
Test.	2,198	4,623	8,271	13,138	933	5,062	8,857	7,804	1,806	4,452	6,932	13,763

Table 1: Number of facts for each benchmark and phase (Training, Validation, and Testing)

		Precision			Recall			Accuracy			F1 Score			AUC			Training (s)	
		A	D	M	A	D	M	A	D	M	A	D	M	A	D	M	D	M
FB15K-237	v1	**100.0**	82.2	98.5	25.6	**40.5**	32.7	62.9	65.9	**66.1**	41.1	**54.2**	49.1	62.0	**68.6**	65.7	15,540	9,192
	v2	**100.0**	85.3	99.5	47.9	**48.5**	41.2	**74.0**	70.0	70.5	**64.5**	61.9	58.3	73.8	**76.0**	70.4	19,080	23,523
	v3	**100.0**	85.6	99.0	43.6	**45.2**	34.6	**71.2**	68.8	67.1	**60.7**	59.2	51.2	71.7	**73.3**	67.4	37,776	45,104
	v4	**100.0**	88.8	99.4	**46.0**	41.6	35.1	**73.0**	68.2	67.5	**63.0**	56.7	52.0	73.0	**73.7**	67.3	87,000	42,540
NELL-995	v1	**97.5**	94.7	96.4	77.0	18.0	**80.0**	87.5	58.5	**88.5**	86.0	30.3	**87.4**	85.7	52.6	**89.6**	402	662
	v2	**100.0**	80.7	**100.0**	**53.1**	41.2	48.3	**76.6**	65.7	74.2	**69.4**	54.5	65.2	**76.3**	74.6	74.2	6,354	3,575
	v3	**100.0**	85.8	**100.0**	47.3	45.6	**55.3**	73.7	69.0	**77.6**	64.2	59.6	**71.2**	69.3	75.3	**77.6**	36,090	13,456
	v4	**100.0**	80.3	99.5	44.3	26.8	**57.2**	72.2	60.1	**78.5**	61.4	40.2	**72.6**	71.6	67.1	**78.5**	3,990	16,839
WN18RR	v1	99.1	97.9	**100.0**	58.5	**73.4**	62.8	79.0	**85.9**	81.4	73.6	**83.4**	77.1	78.8	**92.5**	80.1	312	292
	v2	**100.0**	96.3	**100.0**	**75.7**	69.8	60.8	**87.9**	83.6	80.4	**86.2**	80.9	75.6	59.8	**87.4**	80.3	2,856	856
	v3	99.7	91.3	**100.0**	48.4	**59.7**	28.3	74.1	**77.0**	64.1	65.2	**72.2**	44.1	60.6	**85.2**	64.0	9,984	2,423
	v4	99.9	98.3	**100.0**	**71.6**	65.6	58.0	**85.8**	82.2	79.0	**83.4**	78.7	73.4	59.0	**93.9**	79.0	1,638	409

Table 2: Results for AnyBURL (A), DRUM (D), and our MGNN-based system (M), in percentages

complete dataset that is expected to be completed with dataset $\mathcal{S}_C$. Thus, we took $(\mathcal{S}_I, \alpha)$ for each fact $\alpha \in \mathcal{S}_C$ as a positive example for testing. Moreover, we randomly sampled $|\mathcal{S}_C|$ distinct facts $\beta_1, \ldots, \beta_n$ over the predicates of S_I satisfying $\beta_i \notin \mathcal{S}_C$, and we took each $(\mathcal{S}_I, \beta_i)$ as a negative example for testing. Sampling was necessary because the set of possible facts not contained in $\mathcal{S}_C$ is very large; moreover, sampling at random ensures that our evaluation does not favour a specific way of constructing negative examples during training. Finally, the number of positive and negative examples was the same to ensure that classification metrics are balanced. For each system, we classified the positive and negative testing examples and computed the precision, recall, accuracy, and F1 score of classification in the usual way, and we also computed the *area under the precision-recall curve (AUC)* by considering different classification thresholds between 0.01 and 0.99.

Training. Our system is trained as a denoising autoencoder (Vincent et al., 2010). The training dataset $\mathcal{T}$ is first split with a 9:1 ratio into an incomplete dataset $\mathcal{T}_I$ and a set $\mathcal{T}_C$ of facts that should hold in the completion of $\mathcal{T}_I$. To train our model, we used $(\mathcal{T}_I, \alpha)$ for each fact $\alpha \in \mathcal{T}_C$ as a positive example, and we obtained negative examples $(\mathcal{T}_I, \beta)$ by sampling β in a way analogous to how this was done for testing. We used cross-entropy loss, and we trained our models using Adam optimisation with the standard learning rate (0.01) and weight decay (5×10^{-4}), and a maximum of $50,000$ epochs for training. We implemented the procedure in Pytorch Geometric v1.5.0, and we ran it on a laptop running macOS 10.15.7 with 8 GB of RAM and an Intel Core i5 2.30 GHz CPU.

We trained the baseline systems using their (publicly available) code with their default configurations. For each system and benchmark, we computed the accuracy on the validation dataset for a range of thresholds, and we used in our tests the threshold that maximises accuracy.

Evaluation results. Table 2 shows the results of our evaluation. As one can see, our system was overall very competitive, and it outperformed both baselines on most benchmarks based on NELL-995. Table 2 also shows the training times for DRUM and our system, which are are comparable; no times can be shown for AnyBURL because this system does not train a model.

Similarly to AnyBURL, our system attained extremely high precision scores; however, recall values were significantly lower, especially for the benchmarks based on FB15K. This can be explained by the fact that, after randomly splitting the testing dataset $\mathcal{S}$ into subsets $\mathcal{S}_I$ and $\mathcal{S}_C$, numerous facts of the form $R(a, b)$ in $\mathcal{S}_C$ use pairs of constants a and b that do not occur together in any fact in $\mathcal{S}_I$. By Theorem 10 from Section 3, our transformation can never produce such facts in the output. To confirm our hypothesis that this is the main reason for low recall scores, we additionally split the

Benchmark		Precision	Recall	Accuracy	F1 Score	AUC
FB15K-237	v1	95.5	74.0	85.3	83.4	89.5
	v2	90.3	70.0	79.9	76.9	85.8
	v3	88.8	72.6	81.7	79.9	86.9
	v4	90.5	79.7	85.6	84.7	91.7
NELL-995	v1	92.3	98.8	95.2	95.5	93.3
	v2	83.5	57.3	73.0	68.0	79.4
	v3	82.3	71.1	77.9	76.3	89.0
	v4	78.5	62.4	72.7	69.5	75.8
WN18RR	v1	82.1	100.0	89.1	90.2	95.1
	v2	91.7	98.0	94.6	94.7	98.7
	v3	88.2	98.9	92.9	93.3	89.0
	v4	83.8	99.0	89.9	90.8	97.7

Table 3: Evaluation of the MGNN transformation with alternative splitting of the testing dataset

Benchmark		# Rules captured containing...	
		1 body atom	2 body atoms
FB15K-237	v1	2,102	41,341
	v2	3,370	27,900
	v3	3,199	42,590
	v4	8,464	82,003
NELL-995	v1	149	215
	v2	1,784	3,087
	v3	2,308	5,085
	v4	1,308	4,348
WN18RR	v1	19	3
	v2	22	35
	v3	23	19
	v4	20	54

Table 4: Rules captured by $T_{\mathcal{M}}$

Benchmark		Total AnyBURL	Tree-like AnyBURL	Captured by $T_{\mathcal{M}}$	Captured on $\mathcal{S}_I$
FB15K-237	v1	2812	252	72	1371
	v2	2823	404	102	1048
	v3	4805	490	118	1749
	v4	3574	584	134	1046
NELL-995	v1	891	51	49	707
	v2	1123	181	112	650
	v3	1104	277	181	603
	v4	841	187	108	606
WN18RR	v1	397	15	11	30
	v2	522	21	14	35
	v3	424	21	21	48
	v4	511	16	12	23

Table 5: Comparison of rule extraction with AnyBURL

testing dataset $\mathcal{S}$ so that a fact $R(a, b)$ is placed into $\mathcal{S}_C$ only if $\mathcal{S}_I$ contains a fact of the form $S(a, b)$ or $S(b, a)$ for some predicate S. The performance metrics obtained in this way are shown in Table 3. As one can see, precision scores remain high, whereas recall values increase significantly; as a consequence, accuracy, F1 score, and AUC values improve accordingly.

Extracting Datalog rules. By Definition 9, program $\mathcal{P}_{\mathcal{M}}$ characterising an MGNN $\mathcal{M}$ of dimension k and L layers can be constructed by generating each $(3k, L)$-tree-like rule r over the fixed predicates and checking whether r is captured by $T_{\mathcal{M}}$ using Proposition 7. To reduce the search space, we considered only rules with at most two body atoms. Table 4 shows the numbers of such rules learned on each benchmark.

By analysing the rules, we noted that our transformation learned many rules describing sound inferences such as symmetry (e.g., $sibling(x, y) \leftarrow sibling(y, x)$ on FB15K-237), inverse relations (e.g., $capitalOfArea(x, y) \leftarrow areaHasCapital(y, x)$ on FB15K-237), and relationship subsumption (e.g., $athletePlayedForTeam(x, y) \leftarrow athleteLedSportsTeam(x, y)$ on NELL995).

In addition, we generated rules using AnyBURL on each benchmark and checked how many of those are captured by our transformation;[2] Table 5 summarises our results. The first column provides the total number of rules generated by AnyBURL on each benchmark. The second column indicates how many of those rules were tree-like as per Definition 9. The third column shows how many rules from the second column are captured by our transformation per Definition 6. Finally, the fourth column shows the number of AnyBURL rules r that are captured by $T_{\mathcal{M}}$ *on the specific dataset*—that is, such that $T_r(\mathcal{S}_I) \subseteq T_{\mathcal{M}}(\mathcal{S}_I)$. As one can see, a significant proportion of tree-like rules generated by AnyBURL are captured by $T_{\mathcal{M}}$. Additionally, a significant proportion of (not necessarily tree-shaped) rules generated by AnyBURL are captured by $T_{\mathcal{M}}$ on the specific dataset, which partially explains why many of the predictions by AnyBURL are also obtained by $T_{\mathcal{M}}$.

[2]We could not do this for DRUM since the version of the system available online cannot produce the rules.

5 RELATED WORK

GNN-based approaches to KG completion (Hamaguchi et al., 2017; Schlichtkrull et al., 2018; Xu et al., 2020; Teru et al., 2020; Zhang et al., 2020) are becoming increasingly popular as they are more generally applicable than earlier solutions based on graph embeddings (Rossi et al., 2021). Typically, the predictions made by these techniques cannot be explained symbolically.

Rule mining techniques for KGs can categorised in two groups. Systems in the first category learn a model on training data and provide ways to extract rules from it: DRUM and similar approaches (Yang et al., 2017; Sadeghian et al., 2019; Wang et al., 2020) use RNNs, while others use reinforcement learning (Xiong et al., 2017; Das et al., 2018), dynamic neural module networks (Rocktäschel & Riedel, 2017; Campero et al., 2018), or graph embeddings (Omran et al., 2018; Zhang et al., 2019). However, the formal relationship between the model and the rules extracted from it is unspecified in these approaches; hence, model predictions can differ from the result of applying the rules. Systems in the second category, such as AnyBURL, use training data to identify patterns of paths in the graph that are represented as rules (Galárraga et al., 2015; Meilicke et al., 2019; Ahmadi et al., 2020; Gu et al., 2020); hence, they do not involve training an ML model.

Inductive Logic Programming (ILP) generates rules given a dataset and examples of positive and negative inferences so that rule application yields all positive and no negative examples (Muggleton, 1991). Traditional ILP techniques cannot handle noisy examples, and this limitation also applies to some newer rule learning approaches (Cropper & Muggleton, 2014; Si et al., 2019; Raghothaman et al., 2020). Other recent ILP systems, such as ∂ILP (Evans & Grefenstette, 2018), achieve robustness to noise by interpreting the ILP task as a binary classification problem and providing a differentiable implementation of deduction. These approaches are similar to the rule mining techniques described above, and they compare to our work analogously. Furthermore, they focus on learning rules from small datasets and have been found to struggle with large-scale KGs.

Neuro-symbolic computation focuses on integrating logic with neural networks (d'Avila Garcez et al., 2002). Early rule extraction methods from neural networks considered only propositional (i.e., variable-free) rules (Andrews et al., 1995). Newer approaches focus on identifying the classes of first-order rules for which the immediate consequence operator can be simulated by recurrent (Hölldobler et al., 1999), fibring (Bader et al., 2005), or feed-forward networks (Bader et al., 2007). However, their embedding methods are very different from ours: they transform a dataset into a vector of real numbers by encoding groups of facts as components of the vector, and they can still only extract propositional rules from trained networks. Recently, Dong et al. (2019) presented a network architecture that simulates application of arbitrary function-free first-order rules to a dataset. To this end, they use multi-layer perceptrons to learn Boolean operations between ground atoms and combinatorial tensor calculus to simulate quantification. This work, however, does not provide an algorithm extracting the rules from a trained model.

Existing methods for *explaining the predictions of a GNN* focus on identifying parts of the graph in the embedding space that are most relevant to a given prediction (Ying et al., 2019; Luo et al., 2020; Lin et al., 2021). These techniques are largely independent from the specifics of the model used. In contrast, we focus on a specific type of GNN and provide a logical proof for each prediction.

GNNs were shown capable of expressing graph property tests such as isomorphism, existence of cliques, bipartiteness, and planarity (Xu et al., 2019; Morris et al., 2019; Garg et al., 2020). Moreover, first-order logic queries expressible by GNNs correspond to the description logic $\mathcal{ALCQ}$ (Barceló et al., 2020). Our work follows this line of research by relating GNNs and Datalog.

6 CONCLUSION AND FUTURE WORK

We presented a GNN-based transformation of datasets that mimics a round of application of Datalog rules. The predictions made by our transformation can be explained symbolically, and we have shown our approach to be practically feasible. We see many avenues for future work. On the theoretical side, we shall develop extensions that capture more complex rules (e.g., transitivity), as well as nonmonotonic extensions such as negation or aggregation. On the practical side, we shall establish links with the existing approaches to rule mining and ILP, thus allowing us to compare the performance of all of these heterogeneous approaches in a unified way.

REFERENCES

Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, Reading, MA, 1995.

Charu C. Aggarwal. *Recommender Systems - The Textbook*. Springer, 2016.

Naser Ahmadi, Viet-Phi Huynh, Venkata Vamsikrishna Meduri, Stefano Ortona, and Paolo Papotti. Mining expressive rules in knowledge graphs. *ACM J. Data Inf. Qual.*, 12(2):8:1–8:27, 2020.

Robert Andrews, Joachim Diederich, and Alan B. Tickle. Survey and Critique of Techniques for Extracting Rules from Trained Artificial Neural Networks. *Knowl. Based Syst.*, 8(6):373–389, 1995.

Sebastian Bader, Artur S. d'Avila Garcez, and Pascal Hitzler. Computing First-Order Logic Programs by Fibring Artificial Neural Networks. In *FLAIRS*, pp. 314–319, 2005.

Sebastian Bader, Pascal Hitzler, Steffen Hölldobler, and Andreas Witzel. A Fully Connectionist Model Generator for Covered First-Order Logic Programs. In *IJCAI*, pp. 666–671, 2007.

Pablo Barceló, Egor V. Kostylev, Mikaël Monet, Jorge Pérez, Juan L. Reutter, and Juan Pablo Silva. The Logical Expressiveness of Graph Neural Networks. In *ICLR*. OpenReview.net, 2020.

Antoine Bordes, Nicolas Usunier, Alberto García-Durán, Jason Weston, and Oksana Yakhnenko. Translating embeddings for modeling multi-relational data. In Christopher J. C. Burges, Léon Bottou, Zoubin Ghahramani, and Kilian Q. Weinberger (eds.), *NeurIPS*, pp. 2787–2795, 2013.

Andres Campero, Aldo Pareja, Tim Klinger, Josh Tenenbaum, and Sebastian Riedel. Logical Rule Induction and Theory Learning Using Neural Theorem Proving. *CoRR*, abs/1809.02193, 2018.

Stefano Ceri, Georg Gottlob, and Letizia Tanca. What you always wanted to know about datalog (and never dared to ask). *IEEE Trans. Knowl. Data Eng.*, 1(1):146–166, 1989.

Andrew Cropper and Stephen H. Muggleton. Logical minimisation of meta-rules within meta-interpretive learning. In *ILP*, volume 9046 of *LNCS*, pp. 62–75. Springer, 2014.

Rajarshi Das, Shehzaad Dhuliawala, Manzil Zaheer, Luke Vilnis, Ishan Durugkar, Akshay Krishnamurthy, Alex Smola, and Andew McCallum. Go for a Walk and Arrive at the Answer: Reasoning Over Paths in Knowledge Bases using Reinforcement Learning. In *ICLR*. OpenReview.net, 2018.

Artur S. d'Avila Garcez, Krysia Broda, and Dov M. Gabbay. *Neural-symbolic Learning Systems—Foundations and Applications*. Springer-Verlag, London, 2002.

Tim Dettmers, Pasquale Minervini, Pontus Stenetorp, and Sebastian Riedel. Convolutional 2D knowledge graph embeddings. In Sheila A. McIlraith and Kilian Q. Weinberger (eds.), *AAAI*, pp. 1811–1818. AAAI Press, 2018.

Honghua Dong, Jiayuan Mao, Tian Lin, Chong Wang, Lihong Li, and Denny Zhou. Neural logic machines. In *ICLR*. OpenReview.net, 2019.

Richard Evans and Edward Grefenstette. Learning Explanatory Rules from Noisy Data. *JAIR*, 61: 1–64, 2018.

Luis Galárraga, Christina Teflioudi, Katja Hose, and Fabian M. Suchanek. Fast Rule Mining in Ontological Knowledge Bases with AMIE+. *VLDB J.*, 24(6):707–730, 2015.

V. K. Garg, S. Jegelka, and T. S. Jaakkola. Generalization and Representational Limits of Graph Neural Networks. In *ICML*, pp. 3419–3430, 2020.

Yulong Gu, Yu Guan, and Paolo Missier. Efficient Rule Learning with Template Saturation for Knowledge Graph Completion. *CoRR*, abs/2003.06071, 2020.

Takuo Hamaguchi, Hidekazu Oiwa, Masashi Shimbo, and Yuji Matsumoto. Knowledge transfer for out-of-knowledge-base entities : A graph neural network approach. In *IJCAI*, pp. 1802–1808. AAAI Press, 2017.

William L. Hamilton, Rex Ying, and Jure Leskovec. Representation learning on graphs: Methods and applications. *IEEE Data Eng. Bull.*, 40(3):52–74, 2017.

Aidan Hogan, Eva Blomqvist, Michael Cochez, Claudia d'Amato, Gerard de Melo, Claudio Gutiérrez, Sabrina Kirrane, José Emilio Labra Gayo, Roberto Navigli, Sebastian Neumaier, Axel-Cyrille Ngonga Ngomo, Axel Polleres, Sabbir M. Rashid, Anisa Rula, Lukas Schmelzeisen, Juan F. Sequeda, Steffen Staab, and Antoine Zimmermann. Knowledge graphs. *ACM Comput. Surv.*, 54(4):71:1–71:37, 2021.

Steffen Hölldobler, Yvonne Kalinke, and Hans-Peter Störr. Approximating the Semantics of Logic Programs by Recurrent Neural Networks. *Appl. Intell.*, 11(1):45–58, 1999.

Wanyu Lin, Hao Lan, and Baochun Li. Generative causal explanations for graph neural networks. In *ICML*, volume 139, pp. 6666–6679, 2021.

Zhiyuan Liu and Jie Zhou. *Introduction to Graph Neural Networks*. Morgan and Claypool Publishers, 2020.

Dongsheng Luo, Wei Cheng, Dongkuan Xu, Wenchao Yu, Bo Zong, Haifeng Chen, and Xiang Zhang. Parameterized explainer for graph neural network. In *NeurIPS*, 2020.

Christian Meilicke, Melisachew Wudage Chekol, Daniel Ruffinelli, and Heiner Stuckenschmidt. Anytime bottom-up rule learning for knowledge graph completion. In Sarit Kraus (ed.), *IJCAI*, pp. 3137–3143, 2019.

C. Morris, M. Ritzert, M. Fey, W/ L. Hamilton, J. E. Lenssen, G. Rattan, and M. Grohe. Weisfeiler and Leman Go Neural: Higher-Order Graph Neural Networks. In *AAAI*, pp. 4602–4609, 2019.

Stephen Muggleton. Inductive Logic Programming. *New Gener. Comput.*, 8(4):295–318, 1991.

Pouya G. Omran, Kewen Wang, and Zhe Wang. Scalable Rule Learning via Learning Representation. In *IJCAI*, pp. 2149–2155, 2018.

Mukund Raghothaman, Jonathan Mendelson, David Zhao, Mayur Naik, and Bernhard Scholz. Provenance-guided synthesis of datalog programs. *Proc. ACM Program. Lang.*, 4(62):1–27, 2020.

Tim Rocktäschel and Sebastian Riedel. End-to-end differentiable proving. In *NeurIPS*, pp. 3788–3800, 2017.

Andrea Rossi, Denilson Barbosa, Donatella Firmani, Antonio Matinata, and Paolo Merialdo. Knowledge graph embedding for link prediction: A comparative analysis. *ACM Trans. Knowl. Discov. Data*, 15(2):14:1–14:49, 2021.

Ali Sadeghian, Mohammadreza Armandpour, Patrick Ding, and Daisy Zhe Wang. DRUM: End-To-End differentiable rule mining on knowledge graphs. In *NeurIPS*, pp. 15321–15331, 2019.

Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The Graph Neural Network Model. *IEEE TNNLS*, 20(1):61–80, 2009.

Michael Schlichtkrull, Thomas N. Kipf, Peter Bloem, Rianne Van Den Berg, Ivan Titov, and Max Welling. Modeling relational data with graph convolutional networks. In *ESWC*, pp. 593–607, 2018.

Xujie Si, Mukund Raghothaman, Kihong Heo, and Mayur Naik. Synthesizing datalog programs using numerical relaxation. In *IJCAI*, pp. 6117–6124. ijcai.org, 2019.

Komal K. Teru, Etienne Denis, and Will Hamilton. Inductive relation prediction by subgraph reasoning. In *ICML*, volume 119 of *Proceedings of Machine Learning Research*, pp. 9448–9457. PMLR, 2020.

Rianne van den Berg, Thomas N. Kipf, and Max Welling. Graph convolutional matrix completion, 2017.

Pascal Vincent, Hugo Larochelle, Isabelle Lajoie, Yoshua Bengio, Pierre-Antoine Manzagol, and Léon Bottou. Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion. *J. of Machine Learning Res.*, 11(12), 2010.

Hongwei Wang, Fuzheng Zhang, Jialin Wang, Miao Zhao, Wenjie Li, Xing Xie, and Minyi Guo. Ripplenet: Propagating user preferences on the knowledge graph for recommender systems. In *CIKM*, pp. 417–426, 2018.

Hongwei Wang, Miao Zhao, Xing Xie, Wenjie Li, and Minyi Guo. Knowledge graph convolutional networks for recommender systems. In *WWW*, pp. 3307–3313, 2019a.

Peifeng Wang, Jialong Han, Chenliang Li, and Rong Pan. Logic attention based neighborhood aggregation for inductive knowledge graph embedding. In *AAAI*, pp. 7152–7159, 2019b.

Po-Wei Wang, Daria Stepanova, Csaba Domokos, and J. Zico Kolter. Differentiable learning of numerical rules in knowledge graphs. In *ICLR*. OpenReview.net, 2020.

Wenhan Xiong, Thien Hoang, and William Yang Wang. Deeppath: A reinforcement learning method for knowledge graph reasoning. In *EMNLP*, pp. 564–573. Association for Computational Linguistics, 2017.

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

Xiaoran Xu, Wei Feng, Yunsheng Jiang, Xiaohui Xie, Zhiqing Sun, and Zhi-Hong Deng. Dynamically Pruned Message Passing Networks for Large-scale Knowledge Graph Reasoning. In *ICLR*. OpenReview.net, 2020.

Fan Yang, Zhilin Yang, and William W. Cohen. Differentiable Learning of Logical Rules for Knowledge Base Reasoning. In *NeurIPS*, pp. 2319–2328, 2017.

Zhitao Ying, Dylan Bourgeois, Jiaxuan You, Marinka Zitnik, and Jure Leskovec. GNNExplainer: Generating explanations for graph neural networks. In *NeurIPS*, pp. 9240–9251, 2019.

Fuzheng Zhang, Nicholas Jing Yuan, Defu Lian, Xing Xie, and Wei-Ying Ma. Collaborative knowledge base embedding for recommender systems. In *SIGKDD*, pp. 353–362, 2016.

Wen Zhang, Bibek Paudel, Liang Wang, Jiaoyan Chen, Hai Zhu, Wei Zhang, Abraham Bernstein, and Huajun Chen. Iteratively Learning Embeddings and Rules for Knowledge Graph Reasoning. In *WWW*, pp. 2366–2377, 2019.

Yuyu Zhang, Xinshi Chen, Yuan Yang, Arun Ramamurthy, Bo Li, Yuan Qi, and Le Song. Efficient probabilistic logic reasoning with graph neural networks. In *ICLR*. OpenReview.net, 2020.

A Proof of Proposition 7

Proposition 7. *Let T be an operator on datasets, let r be a constant-free rule of form* (2)*, let μ be a substitution that maps each variable of r to a distinct constant, and let $D_r = \{B_1\mu, \ldots, B_m\mu\}$. If T is monotonic under homomorphisms and $H\mu \in T(D_r)$, then T captures r.*

Proof. Consider T, r, D_r, and μ that satisfy the prerequisites of the proposition. Moreover, consider an arbitrary dataset D and an arbitrary fact $\alpha \in T_r(D)$; we show that $\alpha \in T(D)$. By the definition of operator T_r, there exists a substitution ν such that $\{B_1\nu, \ldots, B_m\nu\} \subseteq D$ and $H\nu = \alpha$. Now let h be the mapping of constants to constants such that $h(x\mu) = x\nu$ for each variable x in occurring in r. This definition of h is correct since μ maps each variable in r to a distinct constant. Also, rule r is constant-free, so $h(D_r) = h(\{B_1\mu, \ldots, B_m\mu\}) = \{B_1\nu, \ldots, B_m\nu\} \subseteq D$ holds; moreover, each constant in D_r is obtained by instantiating a variable of r, so h is defined on all of $\mathsf{cn}(D_r)$). In other words, h is a homomorphism from D_r to D. Since T is monotonic under homomorphisms by the assumption of the proposition, h is also a homomorphism from $T(D_r)$ to $T(D)$. Moreover, $H\mu \in T(D_r)$ holds by our assumption, so the definition of homomorphisms ensures $h(H\mu) \in T(D)$. Finally, atom H is constant-free, which together with the definition of h ensures $h(H\mu) = H\nu$. Thus, we have $H\nu \in T(D)$, as required. □

B PROOF OF THEOREM 8

We prove Theorem 8 in two steps. First, we first introduce in Definition B1 a notion of homomorphisms on (Col, δ)-graphs that is analogous to the notion of homomorphisms on datasets. Then, in Lemma B2 we show that the transformation of (Col, δ)-graphs by an MGNN preserves such homomorphisms. Theorem 8 finally combines this result with our encoding and decoding steps. To formalise these results, we extend the order $\leq$ on real numbers to a partial order on vectors so that $\mathbf{a}_1 \leq \mathbf{a}_2$ holds whenever $(\mathbf{a}_1)_i \leq (\mathbf{a}_2)_i$ holds for each position i.

Definition B1. *A* label-respecting homomorphism *from a* (Col, δ)*-graph* $\langle V, \{E_c\}_{c \in \mathsf{Col}}, \lambda \rangle$ *to a* (Col, δ)*-graph* $\langle V', \{E'_c\}_{c \in \mathsf{Col}}, \lambda' \rangle$ *is a mapping* $h : V \to V'$ *such that* $\lambda(v) \leq \lambda'(h(v))$ *for each vertex* $v \in V$*, and, for each colour* $c \in \mathsf{Col}$ *and each edge in* E_c *between vertices* v *and* w*, vertices* $h(v)$ *and* $h(w)$ *are connected by an edge in* E'_c*.*

Lemma B2. *For each* (Col, δ)*-MGNN* $\mathcal{M}$ *and all* (Col, δ)*-graphs* G *and* G'*, each label-respecting homomorphism* h *from* G *to* G' *is also a label-respecting homomorphism from* $\mathcal{M}(G)$ *to* $\mathcal{M}(G')$*.*

Proof. Consider a (Col, δ)-MGNN $\mathcal{M}$ with components as specified in Definition 3, (Col, δ)-graphs $G = \langle V, \{E_c\}_{c \in \mathsf{Col}}, \lambda \rangle$ and $G' = \langle V', \{E'_c\}_{c \in \mathsf{Col}}, \lambda' \rangle$, and a label-respecting homomorphism h from G to G'. Since $\mathcal{M}(G)$ and $\mathcal{M}(G')$ have the same edges as G and G', respectively, h satisfies the first condition of a label-preserving homomorphism from $\mathcal{M}(G)$ to $\mathcal{M}(G')$. We show that h also satisfies the second condition. To this end, let $\lambda^0, \ldots, \lambda^L$ and $\lambda'^0, \ldots, \lambda'^L$ be the sequences of vertex labelling functions used in the computation of $\mathcal{M}(G)$ and $\mathcal{M}(G')$, respectively. We next show by induction on $0 \leq \ell \leq L$ that $\lambda^\ell(v) \leq \lambda'^\ell(h(v))$ holds for each vertex $v \in V_1$; since classification function cls is monotonically increasing, this will imply that $\mathsf{cls}(\lambda^L(v)) \leq \mathsf{cls}(\lambda'^L(h(v)))$ holds for each vertex $v \in V_1$ as well, which proves our claim.

The base case for $\ell = 0$ holds immediately since $\lambda^0 = \lambda$ and $\lambda'^0 = \lambda'$. For the inductive step, we assume that, for some $\ell - 1$, we have $\lambda^{\ell-1}(v) \leq \lambda'^{\ell-1}(h(v))$ for each $v \in V_1$, and we show that $\lambda^\ell(v) \leq \lambda'^\ell(h(v))$ also holds for each $v \in V_1$. To this end, consider arbitrary i between 1 and the dimension of $\mathbf{b}^\ell$. By Definition 3, the i-th components of $\lambda^\ell(v)$ and $\lambda'^\ell(h(v))$ are equal to $\sigma(x_1)$ and $\sigma(x_2)$, where x_1 and x_2 are as in (3) and (4) for $(\mathbf{A}^\ell)_i$ and $(\mathbf{B}^\ell_c)_i$ the i-th rows of $\mathbf{A}^\ell$ and $\mathbf{B}^\ell_c$, respectively:

$$x_1 = (\mathbf{A}^\ell)_i \, \lambda^{\ell-1}(v) \quad + \sum_{c \in \mathsf{Col}} (\mathbf{B}^\ell_c)_i \max\{\lambda^{\ell-1}(w) \mid w \in N_c(v)\} \quad + (\mathbf{b}^\ell)_i, \tag{3}$$

$$x_2 = (\mathbf{A}^\ell)_i \, \lambda'^{\ell-1}(h(v)) + \sum_{c \in \mathsf{Col}} (\mathbf{B}^\ell_c)_i \max\{\lambda'^{\ell-1}(u) \mid u \in N_c(h(v))\} + (\mathbf{b}^\ell)_i. \tag{4}$$

By the inductive hypothesis, we have $\lambda^{\ell-1}(v) \leq \lambda'^{\ell-1}(h(v))$; moreover, all numbers in $\mathbf{A}^\ell$ are nonnegative, so we clearly have $(\mathbf{A}^\ell)_i \lambda^{\ell-1}(v) \leq (\mathbf{A}^\ell)_i \lambda'^{\ell-1}(h(v))$. In addition, mapping h satisfies the first property of label-preserving homomorphisms so, for each colour $c \in \mathsf{Col}$ and each $w \in N_c(v)$, we have $h(w) \in N_c(h(v))$; moreover, by the inductive hypothesis, we have $\lambda^{\ell-1}(w) \leq \lambda'^{\ell-1}(h(w))$. By combining these two observations, we have

$$\max\{\lambda^{\ell-1}(w) \mid w \in N_c(v)\} \leq \max\{\lambda'^{\ell-1}(u) \mid u \in N_c(h(v))\},$$

which, together with the fact that all numbers in $(\mathbf{B}^\ell_c)_i$ are nonnegative, implies

$$(\mathbf{B}^\ell_c)_i \max\{\lambda^{\ell-1}(w) \mid w \in N_c(v)\} \leq (\mathbf{B}^\ell_c)_i \max\{\lambda'^{\ell-1}(u) \mid u \in N_c(h(v))\}.$$

Consequently, we have $x_1 \leq x_2$. Finally, activation function σ is monotonically increasing, so $\sigma(x_1) \leq \sigma(x_2)$ holds, which completes our proof of the inductive step. □

Theorem 8. *For each* (Col, δ)*-MGNN* $\mathcal{M}$*, mapping* $T_\mathcal{M}$ *is monotonic under homomorphisms.*

Proof. Consider an arbitrary (Col, δ)-MGNN $\mathcal{M}$, arbitrary datasets D and D', and an arbitrary homomorphism h from D to D'; thus, $h(D) \subseteq D'$ holds. Now let G_D and $G_{D'}$ be (Col, δ)-graphs obtained by transforming D and D' as in Definition 2. Moreover, let g be the mapping from the vertices of G_1 to the vertices of G_2 defined as follows:

- $g(v_a) = v_{h(a)}$ for each vertex v_a of G_1; and

- $g(v_{a,b}) = v_{h(a),h(b)}$ for each vertex $v_{a,b}$ of G_1.

It is straightforward to see that g is a label-respecting homomorphism from G_D to $G_{D'}$. By Lemma B2, mapping g is also a label-respecting homomorphism from $\mathcal{M}(G_D)$ to $\mathcal{M}(G_{D'})$. Thus, however, implies that h is also a homomorphism from $T(D)$ to $T(D')$, as required. □

C Proof of Theorem 10

Theorem 10. *For each* (Col, δ)*-MGNN* $\mathcal{M}$ *and each dataset* D*, we have* $T_{\mathcal{M}}(D) = T_{\mathcal{P}_{\mathcal{M}}}(D)$.

Proof. Consider arbitrary $\mathcal{M}$, $\mathcal{P}_{\mathcal{M}}$, and D as stated in the claim. Let L be the number of layers of $\mathcal{M}$, and let k be the dimension of $\mathcal{M}$. Since operator $T_{\mathcal{P}_{\mathcal{M}}}$ captures each rule in $\mathcal{P}_{\mathcal{M}}$ by definition, we clearly have $T_{\mathcal{P}_{\mathcal{M}}}(D) \subseteq T_{\mathcal{M}}(D)$. For the converse, consider an arbitrary fact $\alpha \in T_{\mathcal{M}}(D)$. We next construct a rule r for which we prove $r \in \mathcal{P}_{\mathcal{M}}$ and $\alpha \in T_r(D)$.

Let G_D be the (Col, δ)-graph obtained from D by the transformation in Definition 2, and let V_D be the set of vertices of G_D. For each $\ell \in \{0, \ldots, L\}$, let $\mathbf{v}^\ell$ be the vector labelling each $v \in V_D$ in the computation of $\mathcal{M}(G_D)$. Moreover, let μ be a substitution that maps each variable to a distinct constant. We next construct an atom H, a conjunction C, a substitution ν from the variables in C to $\mathsf{cn}(D)$, a set U of fresh vertices (i.e., vertices not occurring in G_D) of the form u_d and $u_{d,e}$ for d and e constants from the image of μ, and mappings $\xi : U \to V_D$ and $M_c : U \to 2^{V_D}$ for each $c \in \mathsf{Col}$. We also assign each vertex in U to a level between 0 and L, and we identify a single vertex from U as the *root* vertex. In the rest of this proof, we use letters a and b for the constants in D, letters d and e for the constants in the image of μ, possibly indexed letters v, w, and s for the vertices in V_D, and possibly indexed letters u and t for the vertices in U. Our construction is by induction from level L down to level 1, and we prove in parallel the following properties:

P1. for each vertex of the form $u_d \in U$, for x the variable such that $\mu(x) = d$, and for $a = \nu(x)$, we have $\xi(u_d) = v_a$;

P2. for each vertex of the form $u_{d,e} \in U$, for x and y variables such that $\mu(x) = d$ and $\mu(y) = e$, and for $a = \nu(x)$ and $b = \nu(y)$, we have $\xi(u_{d,e}) = v_{a,b}$; and

P3. conjunction C satisfies $\nu(C) \subseteq D$.

We initialise C as the empty conjunction, and we set ν, ξ, and each M_c for $c \in \mathsf{Col}$ as the empty mappings. For the induction base, we consider the form of the atom $\alpha \in T_{\mathcal{M}}(D)$ and proceed as follows; the base case also identifies the root vertex. Properties P1–P3 clearly hold after these steps.

- If α is of the form $A(a)$, then V_D contains vertex v_a. We introduce a fresh variable x, and we let d be the constant such that $d = \mu(x)$. We define $\nu(x) = a$; we define $H = A(x)$; we introduce vertex u_d of level L; we define $\xi(u_d) = v_a$; and we make u_d the root vertex. Finally, we extend C with atom $B(x)$ for each $B(a) \in D$. Note that C can be empty if D does not contain a fact such as $B(a)$. However, $v_a \in V_D$ implies that constant a must occur in D in a fact of the form $S(a, b)$ or $S(b, a)$. Thus, $N_c(v_a) \neq \emptyset$ for $c \in \{c_1, c_2, c_4\}$. Since $L \geq 1$, vertex u_d is processed in the inductive step below, which will extend C with an atom that contains variable x.
- If α is of the form $R(a, b)$, then V_D contains vertices $v_{a,b}$ and $v_{b,a}$; due to the latter, D contains at least one fact of the form $S(a, b)$ or $S(b, a)$, and so V_D contains vertices v_a and v_b. We introduce fresh variables x and y, and we let d and e be constants such that $d = \mu(x)$ and $e = \mu(y)$. We define $\nu(x) = a$ and $\nu(y) = b$; we define $H = R(x, y)$; we introduce vertices u_d, u_e, $u_{d,e}$, and $u_{e,d}$ of level L; we define $\xi(u_d) = v_a$, $\xi(u_e) = v_b$, $\xi(u_{d,e}) = v_{a,b}$, and $\xi(u_{e,d}) = v_{b,a}$; and we make $u_{d,e}$ the root vertex. Moreover, we extend C with an atom $S(x, y)$ for each $S(a, b) \in D$, an atom $S(y, x)$ for each $S(b, a) \in D$, an atom $B(x)$ for each $B(a) \in D$, and an atom $B(y)$ for each $B(b) \in D$; note that, C cannot empty after this step. Finally, we define $M_c(u) = N_c(\xi(u))$ for each $u \in \{u_{d,e}, u_{e,d}\}$ and colour $c \in \mathsf{Col}$; note that $M_{c_1}(u)$, $M_{c_2}(u)$, and $M_{c_3}(u)$ are all singletons, and $M_{c_4}(u) = \emptyset$.

Next, we consider each level ℓ with $1 \leq \ell \leq L$ that has already been processed, and we consider each vertex of the form u_d of level ℓ. Let x be the variable such that $\mu(x) = d$, and let $a = \nu(x)$; property P1 ensures $\xi(u_d) = v_a$. For each colour $c \in \{c_1, c_2, c_4\}$, we choose a smallest set $M_c(u_d) \subseteq N_c(v_a)$ such that, for each $\ell' \in \{0, \ldots, \ell - 1\}$ and each $i \in \{1, \ldots, k\}$, there exists a vertex $w \in M_c(u_d)$ satisfying

$$(\mathbf{w}^{\ell'})_i = \max\{(\mathbf{s}^{\ell'})_i \mid s \in N_c(v_a)\}. \tag{5}$$

Note that set $M_c(u_d)$ may not be unique, but any set satisfying (5) can be selected. There are at most k distinct values of $(\mathbf{w}^{\ell'})_i$ for each ℓ', so set $M_c(u_d)$ contains at most $k(L - \ell)$ elements. We also define $M_{c_3}(u_d) = \emptyset$. We now consider each vertex $v \in M_{c_1}(u_d) \cup M_{c_2}(u_d) \cup M_{c_4}(u_d)$. By Definition 2, such v is of the form $v_{a,b}$, $v_{b,a}$, or v_b for $b \in \mathsf{cn}(D)$, depending on the set from which v is drawn. Set V_D clearly also contains vertices $v_{b,a}$ and v_b if $v \in M_{c_1}(u_d)$, and vertices $v_{a,b}$ and v_b if $v \in M_{c_2}(u_d)$. Moreover, if $v \in M_{c_4}(u_d)$, then V_D clearly contains v_b, and $M_{c_4}(u_d) \subseteq N_{c_4}(v_a)$ ensures that D contains a fact of the form $S(a, b)$ or $S(b, a)$; thus, V_D also contains $v_{a,b}$ and $v_{b,a}$. We introduce a fresh variable y, and we let e be the constant such that $e = \mu(y)$. We define $\nu(y) = b$; we introduce vertices u_e, $u_{d,e}$, and $u_{e,d}$ of level $\ell - 1$; and we define $\xi(u_e) = v_b$, $\xi(u_{d,e}) = v_{a,b}$, and $\xi(u_{e,d}) = v_{b,a}$. Moreover, we extend C with an atom $S(x, y)$ for each $S(a, b) \in D$, an atom $S(y, x)$ for each $S(b, a) \in D$, and an atom $B(y)$ for each $B(b) \in D$. Finally, we define $M_c(u) = N_c(\xi(u))$ for each $u \in \{u_{d,e}, u_{e,d}\}$ and colour $c \in \mathsf{Col}$; note that $M_{c_1}(u)$, $M_{c_2}(u)$, and $M_{c_3}(u)$ are all singletons, and $M_{c_4}(u) = \emptyset$. Since each $M_c(u_d)$ with $c \in \{c_1, c_2, c_4\}$ contains at most $k(L - \ell)$ vertices and we consider three colours, this step introduces at most $3k(L - \ell)$ fresh variables. Moreover, note that each fresh variable y occurs in C in an atom of the form $S(x, y)$ or $S(y, x)$. Finally, it should be clear that P1–P3 hold after level ℓ is processed.

This completes our inductive construction. At this point, either $H = A(x)$ and C is a nonempty $(3k, L)$-tree-like conjunction for x, or $H = R(x, y)$ and $C = C_x \wedge B_{x,y} \wedge C_y$ where C_x and C_y are $(3k, L)$-tree-like conjunctions for x and y, respectively, and $B_{x,y}$ is a nonempty conjunction of atoms of the form $S(x, y)$ or $S(y, x)$. Thus, rule $H \leftarrow C$ is of the form $A(x) \leftarrow C_x$ or $R(x, y) \leftarrow C_x \wedge B_{x,y} \wedge C_y$. To complete the proof, we next show this rule is captured by $T_{\mathcal{M}}$. Then, Definition 9 ensures ensures $H \leftarrow C \in \mathcal{P}_{\mathcal{M}}$; together with $H\nu = \alpha$ and $\nu(C) \subseteq D$, we have $\alpha \in T_{\mathcal{P}_{\mathcal{M}}}(D)$, which proves our claim.

We use Proposition 7 to show that $H \leftarrow C$ is captured by $T_{\mathcal{M}}$. To this end, let $D_C = \{\alpha\mu \mid \alpha \in C\}$, and let G_{D_C} be the (Col, δ)-graph constructed using vertices in U obtained by applying the encoding from Definition 2 to dataset D_C. Finally, let $\mathbf{u}^\ell$ be the labelling associated with each vertex $u \in U$ in the computation of $\mathcal{M}(G_{D_C})$. The construction of C clearly ensures the following two properties for each colour $c \in \mathsf{Col}$ and each vertex $u \in U$:

R1. for each vertex $v \in M_c(u)$, there exists a vertex $t \in N_c(u)$ such that $\xi(t) = v$; and

R2. for each vertex $t \in N_c(u)$ in graph G_{D_C}, we have $\xi(t) \in N_c(\xi(u))$.

We now show by the induction on ℓ that, for each $0 \leq \ell \leq L$ and each vertex $u \in U$ whose level is at least ℓ, we have $\mathbf{u}^\ell = \mathbf{v}^\ell$ for $v = \xi(u)$. For the base case $\ell = 0$, we have the following possibilities.

- Assume u is of the form u_d. Let x be the variable such that $d = \mu(x)$, and let $a = \nu(x)$. Property P1 ensures $\xi(u_d) = v_a$, and the construction of C ensures $B(x) \in C$ if and only if $B(a) \in D$; thus, $\mathbf{u}^0_d = \mathbf{v}^0_a$ holds.
- Assume u is of the form $u_{d,e}$. Let x and y be variables such that $\mu(x) = d$ and $\mu(y) = e$, and let $a = \nu(x)$ and $b = \nu(y)$. Property P2 ensures $\xi(u_{d,e}) = v_{a,b}$, and the construction of C ensures $S(x, y) \in C$ if and only if $S(a, b) \in D$; thus, $\mathbf{u}^0_{d,e} = \mathbf{v}^0_{a,b}$ holds.

For the induction step, assume that the property holds for some $\ell - 1$, and consider an arbitrary vertex $u \in U$ whose level is at least ℓ. Let $v = \xi(u)$. The induction assumption ensures $\mathbf{u}^{\ell-1} = \mathbf{v}^{\ell-1}$. Also, for each colour $c \in \mathsf{Col}$, our construction ensures that each vertex $t \in N_c(u)$ is of level at least $\ell - 1$, so the induction assumption ensures $\mathbf{t}^{\ell-1} = \mathbf{w}^{\ell-1}$ for $w = \xi(t)$. Finally, property R2 ensures that each $\mathbf{t}^{\ell-1}$ used to compute $\mathbf{u}^\ell$ is considered when computing $\mathbf{v}^\ell$, and property R1 ensures that the maximal values of $\mathbf{t}^{\ell-1}$ are also present in the computation of $\mathbf{u}^\ell$. Thus, $\mathbf{u}^\ell = \mathbf{v}^\ell$ holds.

To complete the proof, let u be the root vertex, and let $v = \xi(u)$. The above property ensures $\mathbf{u}^L = \mathbf{v}^L$, and so $\mathsf{cls}(\mathbf{u}^L) = \mathsf{cls}(\mathbf{v}^L)$ as well. Thus, we have $\mu(H) \in T_{\mathcal{M}}(D_C)$, so Proposition 7 allows us to conclude that rule $H \leftarrow C$ is captured by $T_{\mathcal{M}}$. $\square$