EXPLAINABLE GNN-BASED MODELS OVER KNOWLEDGE GRAPHS

David Tena Cucala  
Department of Computer Science  
University of Oxford, UK

Bernardo Cuenca Grau  
Department of Computer Science  
University of Oxford, UK

Egor V. Kostylev  
Department of Informatics  
University of Oslo, Norway

Boris Motik  
Department of Computer Science  
University of Oxford, UK

ABSTRACT

Graph Neural Networks (GNNs) are often used to learn transformations of graph data. While effective in practice, such approaches make predictions via numeric manipulations so their output cannot be easily explained symbolically. 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 rule-based formalism. In particular, 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 automatically extract rules that completely characterise 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) describe an application domain as a collection of entities and their relationships, and they are frequently used in applications such as Web search, recommendation, fraud detection, and data integration. Many such applications involve realising a transformation on knowledge graphs. For example, in a recommender system, we can represent user-item interactions as a graph (Aggarwal, 2016) and extend it with external KGs to improve accuracy, diversity, and interpretability of recommendations (Zhang et al., 2016; Wang et al., 2018; 2019a); the recommender system can then transform this graph into another graph containing recommended user-item interactions. Moreover, a KG completion system (Rossi et al., 2021; Bordes et al., 2013; Teru et al., 2020) transforms an input graph into another graph containing relationships missing in the input. Usually, such transformations are not given explicitly, but are learned from examples.

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

While such approaches are effective in practice, the transformation can be difficult to understand because GNNs compute predictions through numeric calculations. For example, it would be useful to know that a system recommended novel The Idiot because the user liked Crime and Punishment and both novels were written by Dostoevsky. There is a growing interest in explaining GNN predictions. Numerous methods focus on identifying relevant subgraphs in the embedding space (Ying et al., 2019; Luo et al., 2020; Lin et al., 2021), but such approaches do not explain predictions symbolically—that is, by showing how a prediction can be derived from the input KG via logical inferences of a knowledge representation formalism. Datalog (Abiteboul et al., 2011)
is a prominent such formalism. It allows one to model an application domain using ‘if–then’ rules, which can be applied to facts to derive new facts. For example, we can represent knowledge about literature using facts Author(Dostoevsky, Crime and Punishment) and Author(Dostoevsky, The Idiot), and past interactions using fact Likes(Crime and Punishment). Then, applying rule \(Author(x, y_1) \land Author(x, y_2) \land Likes(y_1) \rightarrow Recommend(y_2)\) to these facts derives fact \(Recommend(The Idiot)\). Datalog engines (Motik et al., 2014) can generate human-readable proofs explaining why a fact follows from the rules and the data, which can help foster trust in system predictions, ensure norm compliance, and enable verification of fairness standards.

The main contribution of this paper is a new family of GNN-based transformations of KGs that can be trained from examples as usual, but whose predictions can be explained symbolically using Datalog rules. Our transformation consists of the three stages mentioned earlier, but these were carefully crafted to ensure that the transformation can always be described symbolically. To this end, we introduce a novel class of monotonic GNNs (MGNNS) that ensure an important property: when values in input feature vectors increase, no value of an output feature vector decreases. 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 each MGNN, there exists an equivalent set of rules such that, on each KG, 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 and use it to provide further predictions; however, such predictions can always be explained using rules. For example, we could train an MGNN on a graph containing examples of credit applications and then support decision-making for new applications by applying the model to a different graph. For each MGNN obtained in such a way, there exists a set of rules that fully explains each credit recommendation; extracting all such rules can be computationally expensive, but one can typically extract a representative subset of such rules in practice. Thus, we integrate ML and symbolic AI in a single framework, allowing applications to use the best of both worlds.

We demonstrate the effectiveness of our approach by applying it to classification tasks in KG completion. Using well-known benchmarks, we compare our system’s performance with the state-of-the-art KG completion systems DRUM (Sadeghian et al., 2019) and AnyBURL (Mellicke et al., 2019). We show that, in addition to providing an exact correspondence between the model and the extracted rules, our technique also offers competitive performance.

Our proofs are given in the appendix, and the source code 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 \(a, a_1, a_2\) are constants. Constants represent entities from an application domain, unary predicates typically represent entity types, and binary predicates represent entity relationships. For example, the statements that a user likes Crime and Punishment and it is a novel written by Fyodor Dostoevsky can be represented as facts Likes(Crime and Punishment) and Author(Dostoevsky, Crime and Punishment). KGs in formats such as RDF can be transformed into such a dataset in the obvious way. For a dataset, \(cn(D)\) is the set of constants occurring in \(D\).

Our transformation is realised as an operator \(T_M\) that is parameterised by a GNN \(M\) and that maps datasets to datasets. A dataset \(D\) is first encoded into a graph \(G_D\) whose vertices are labelled with numeric feature vectors; graph \(G_D\) is then processed by \(M\); finally, the result is decoded to the output dataset \(T_M(D)\). In Section 3, we show that, if \(M\) is a special kind of GNN that we call monotonic, then \(T_M\) is equivalent to a round of Datalog rule application, and we can automatically extract from \(T_M\) an equivalent set of rules. The space of encodings of \(D\) consists of coloured graphs, which extend standard undirected graphs in two ways. First, each vertex is labelled by a numeric feature vector of a fixed dimension, which allows the graph to be processed by \(M\). Second, each edge in the graph is assigned a colour, which allows us to distinguish different types of connections.
Figure 1 shows the encoding of the dataset \( D \) from the introduction. We introduce vertices \( v_D \), \( v_{CP} \), and \( v_l \) in \( G_D \) to represent constants \( Dostoevsky \), \( The_Idiot \), and \( Crime_and_Punishment \). To represent relationships between constants, we introduce vertices \( v_{Ds,CP} \), \( v_{CP,Ds} \), \( v_{Ds,I} \), and \( v_{I,Ds} \): we do not introduce vertices \( v_{CP,I} \) and \( v_{I,CP} \) because the corresponding constants do not occur together in a fact of \( D \). We then encode unary and binary predicates using feature vectors labelling the graph’s vertices. To this end, we assign position 1 to predicate \( Likes \) and position 3 to predicate \( Author \). Then, we represent fact \( Likes(Crime_and_Punishment) \) by setting the first element of the feature vector of \( v_{CP} \) to one, and we represent fact \( Author(Dostoevsky, Crime_and_Punishment) \) by setting the third element of the feature vector of \( v_{Ds,CP} \) to one. Finally, we connect with coloured edges all pairs of vertices of \( G_D \) that refer to related constants. For example, we connect vertices \( v_D \) and \( v_{Ds,CP} \) by an edge of colour \( c_1 \) to indicate that \( Dostoevsky \) occurs first in the constant pair of \( v_{Ds,CP} \). We analogously connect \( v_{Ds,CP} \) and \( v_{CP} \) by an edge of colour \( c_2 \). We connect \( v_{CP,Ds} \) and \( v_{Ds,CP} \) by an edge of colour \( c_3 \) to indicate that the constant pairs of the two vertices are inverses of each other. Finally, we connect \( v_{CP} \) and \( v_{Ds} \) by an edge of colour \( c_4 \) to indicate that the two constants occur jointly in a fact of \( D \).

Formally, we fix a set of four colours \( Col = \{c_1, c_2, c_3, c_4\} \). We assume that datasets and derived facts will use finitely many predicates, and we fix a list \( A_1, \ldots, A_n, R_{e+1}, \ldots, R_m \) of available unary and binary predicates. We consider vectors and matrices over the reals \( \mathbb{R} \) and its nonnegative subset \( \mathbb{R}^+ \). For \( \mathbf{v} \) a vector, \( (\mathbf{v})_i \) is the \( i \)-th element of \( \mathbf{v} \). We apply scalar functions to vectors and matrices element-wise; for example, for \( \mathbf{v}_1, \ldots, \mathbf{v}_n \) vectors of equal dimension, \( \max\{\mathbf{v}_1, \ldots, \mathbf{v}_n\} \) is the vector whose \( i \)-th element is \( \max\{(\mathbf{v}_1)_i, \ldots, (\mathbf{v}_n)_i\} \). Finally, a function \( \sigma : \mathbb{R} \rightarrow \mathbb{R} \) is monotonically increasing if \( x < y \) implies \( \sigma(x) \leq \sigma(y) \), and function \( \sigma \) is \( \text{Boolean} \) if it ranges over \( \{0, 1\} \). The following definition describes the kind of graph that our encoding produces.

**Definition 1.** A \((\mathcal{C}ol, \delta)\)-graph is a tuple \( (V, \{E_\ell\}_{\ell \in \mathcal{C}ol}, \lambda) \) where \( V \) is a finite set of vertices; for each \( \ell \in \mathcal{C}ol \), \( E_\ell \) is a set of \( \ell \)-coloured undirected edges (possibly including self-loops) over \( V \); and labelling \( \lambda \) assigns to each \( v \in V \) a vector \( \lambda(v) \) of dimension \( \delta \). If \( \lambda \) is implicitly understood, \( \lambda(v) \) is written as \( \mathbf{v} \). For \( \ell \in \mathcal{C}ol \) and \( v \in V \), \( N_\ell(v) \) is the set of vertices connected to \( v \) by an edge in \( E_\ell \).

We are now ready to define our encoding of an arbitrary dataset \( D \) into a \((\mathcal{C}ol, \delta)\)-graph \( G_D \).

**Definition 2.** The encoding of a dataset \( D \) is the \((\mathcal{C}ol, \delta)\)-graph \( G_D \) that contains a vertex \( v_a \) for each constant \( a \in \text{cn}(D) \), 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 \( v_{a,b} \), labelling \( (v_{a,b})_i \) is 1 if \( A_{i}(a) \in D \) or 0 otherwise. For each \( v_{a,b} \), labelling \( (v_{a,b})_i \) is 1 if \( R_{i}(a, b) \in D \) and 0 otherwise.

Graph \( G_D \) will be processed by a GNN. As we explain in detail in 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 \((\mathcal{C}ol, \delta)\)-monotonic graph neural network (MGNN) \( \mathcal{M} \) with \( L \geq 1 \) layers is a tuple \( (\{A^\ell\}_{1 \leq \ell \leq L}, \{B^\ell\}_{1 \leq \ell \leq L}, \{b^\ell\}_{1 \leq \ell \leq L}, \lambda, \text{cls}) \). For each \( \ell \in \{1, \ldots, L\} \) and each \( \ell \in \mathcal{C}ol \), \( A^\ell \) and \( B^\ell \) are matrices over dimension \( n^\ell \times n^\ell \), and \( b^\ell \) is a vector over \( \mathbb{R} \) of dimension \( n^\ell \) such that \( n^1 = n^L = \delta \), and \( n^\ell = n^{\ell-1} \) for each \( \ell \in \{2, \ldots, L\} \). Moreover, \( \lambda : \mathbb{R}^+ \rightarrow \{0, 1\} \) is a monotonically increasing activation function, and \( \text{cls} : \mathbb{R}^+ \rightarrow \{0, 1\} \) is a monotonically increasing classification function. The dimension of \( \mathcal{M} \) is \( \max\{n^1, \ldots, n^L\} \).
Applying $\mathcal{M}$ to a $(\text{Col}, \delta)$-graph $G = (V; \{E_c\}_{c \in \text{Col}}, \lambda)$ 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

$$v^\ell = \sigma(A^\ell v^{\ell-1} + \sum_{c \in \text{Col}} B^\ell_c \max\{w^{\ell-1} \mid w \in N_c(v)\} + b^\ell).$$  

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

Intuitively, an MGNN consists of $L$ layers, where each layer $\ell \in \{1, \ldots, L\}$ is defined by a matrix $A^\ell$, a matrix $B^\ell_c$ for each colour $c$, and a bias vector $b^\ell$; these constitute the learnable parameters of the model. In addition, an MGNN also uses functions $\sigma$ and $\text{cls}$ which are fixed and explicitly given. An MGNN transforms an arbitrary $(\text{Col}, \delta)$-graph $G$ into another $(\text{Col}, \delta)$-graph $\mathcal{M}(G)$ as follows. For each layer $\ell \in \{0, \ldots, L\}$, each vertex $v$ in $G$ is associated with a labelling $v^\ell$. Labelling $v^\ell$ is the labelling $v$ of $v$ in $G$, and, for each layer $\ell$, vertex $v$ is labelled by a feature vector $v^\ell$ computed as specified in equation (1). The feature vector of $v$ in the resulting graph is computed by applying the classification function $\text{cls}$ to the final labelling $v^L$. Thus, MGNNs specialise GNNs 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 $A^\ell$ and $B^\ell_c$ (but the elements of bias vectors $b^\ell$ 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 (cf. Figure 1). Thus, for each MGNN $\mathcal{M}$, we obtain an operator $T_\mathcal{M}$ that is specified as follows.

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

Note that the dimensions of the matrices of $\mathcal{M}$ are independent from the number of constants in a dataset; in fact, $\mathcal{M}$ can be trained on one dataset and then applied to another dataset that contains the same predicates but completely different constants. Our transformation is thus ideally suited for inductive settings such as KG completion and recommendation (Teru et al., 2020; Hamaguchi et al., 2017; van den Berg et al., 2017), where training and testing can involve distinct entities.

## 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 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 $B_i$.

$$H \leftarrow B_1 \land \cdots \land B_m$$

A (Datalog) *program* is a finite set of rules. A *substitution* $\nu$ is a mapping of finitely many variables to constants; for $\alpha$ an atom, $\alpha\nu$ is the result of replacing in $\alpha$ each variable $x$ on which $\nu$ is defined with $\nu(x)$. Each rule $r$ of form (2) defines the immediate consequence operator $T_r$ on datasets: for $D$ a dataset, set $T_r(D)$ contains fact $H\nu$ for each substitution $\nu$ 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 $\text{cn}(D)$ 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 $P$, operator $T_P$ is monotonic under homomorphisms. For example, let $D = \{A(a), R(a, b)\}$ and $P = \{S(x, y) ← A(x) ∧ R(x, y)\}$, so $T_P(D) = \{S(a, b)\}$. First, operator $T_P$ is monotonic: adding facts to $D$ cannot invalidate a consequence of $T_P$ because the body of $r$ still matches so $r$ derives $S(a, b)$. Second, operator $T_P$ is resilient to replacement of constants. For example, let $D' = \{A(c), R(c, c)\}$ be obtained from $D$ by replacing $a$ and $b$ with $c$. Rule $r$ 'cannot tell' that the constants have been replaced; thus, it derives $T_P(D') = \{S(c, c)\}$, which is obtained from $T_P(D)$ by the same replacement. Monotonicity under homomorphisms combines both properties. We next show that rules can be extracted from any operator satisfying this property.

**Definition 6.** An operator $T$ on datasets captures a rule or program $\alpha$ 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 $\mu$ be a substitution that maps each variable of $r$ to a distinct constant, and let $D_\mu = \{B_1\mu, \ldots, B_m\mu\}$. If $T$ is monotonic under homomorphisms and $H\mu \in T(D_\mu)$, 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, $T$ necessarily 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 $r$: we replace each variable in $r$ with a distinct constant, we collect the resulting body atoms in a dataset $D_r$, we apply $T$ to $D_r$, and we check whether the result contains the replaced head atom of $r$. Clearly, $T$ captures a constant-free program $P$ if $T$ captures each rule $r ∈ P$, so we can just apply this check to each $r ∈ P$ to verify whether $T$ captures $P$. This provides us with the necessary machinery for rule extraction.

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

**Theorem 8.** For each $(\text{Col}, \delta)$-MGNN $M$, mapping $T_M$ is monotonic under homomorphisms.

Theorem 8 is a consequence of the restrictions on MGNNs mentioned 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 some elements of feature vectors of $G_D$ from 0 to 1; moreover, the elements of $A'$ and $B'$ are nonnegative so the arguments of the predicate $A$ in equation (1) can only increase; finally, functions $\sigma$ and $\delta$ are monotonically increasing so the elements of the feature vectors of $M(G_D)$ can only flip from 0 to 1, which is equivalent to deriving more facts in $T_M(D)$. For the resilience of constant merging, replacing, say, constant $b$ with constant $a$ in $D$ is tantamount to merging vertices $v_b$ and $v_a$. Rule $T_M$ can derive more facts. However, if 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 literature), then elements of the feature vectors of $M(G_D)$ could decrease due to merging, which would invalidate Theorem 8.

Theorem 8 tells us that we can check whether $T_M$ captures a rule $r$ using Proposition 7. This is a key step of our rule extraction algorithm. In particular, we next show that it is sufficient to consider only rules of a certain structure, and that the number of such rules is determined by the number of predicates $\delta$, the dimension of $M$, and the number of layers of $M$. Thus, to obtain a program $P_M$ deriving the same consequences as $T_M$, we can, in principle, enumerate all such rules, apply Proposition 7 to each rule, and keep the rules that pass this check. Such a naive procedure can be optimised in practice. For example, we can reduce the number of considered rules by observing that rules can logically imply other rules; for example, if we determine that a rule $B(x) ← A(x)$ is captured, then we do not need to consider any rule of the form $B(x) ← A(x) \land C$ where $C$ is an arbitrary conjunction: each such rule is logically implied by $B(x) ← A(x)$ and is thus redundant.

**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 $v$ 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 $A(x) \leftrightarrow C_x$ or $R(x, y) \leftarrow C_x \land B_{x,y} \land C_y$, 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 $S(y, x)$.\footnote{Note that $C_x$ cannot be empty in a rule of form $A(x) \leftrightarrow C_x$ since $x$ is required to occur in the body.}

For $M$ a $(\text{Col}, \delta)$-MGNN with $L$ layers and dimension $k$ (see Definition\textsuperscript{2}, $P_M$ is the program that contains, up to variable renaming, each $(3k, L)$-tree-like rule that is captured by $T_M$.\footnote{Theorem\textsuperscript{10} also describes the expressiveness (i.e., the kind of possible predictions) of $T_M$. For example, no MGNN can capture rule $r = R(x, z) \leftarrow R(x, y) \land R(y, z)$. To see this is the case, let $D = \{R(a, b), R(b, c)\}$. For an arbitrary MGNN $M$, Theorem\textsuperscript{10} ensures that all predictions of $T_M$ involving binary facts are made by rules of the form $R(x, y) \leftarrow C_x \land B_{x,y} \land C_y$; however, no such rule can derive $R(a, c)$ on $D$, so we necessarily have $R(a, c) \notin T_M(D)$ as well.}

The primal graph of conjunction $A(x) \land R(x, y) \land S(x, z)$ is a tree with root $x$ and children $y$ and $z$; thus, it is (2,1)-tree-like for $x$, but not (1,1)-tree-like for $x$ (since $x$ is at distance 0 from itself it can have at most one child). In contrast, the primal graph of $A(x) \land R(x, y) \land S(x, z) \land T(y, z)$ is a triangle with nodes $x$, $y$, and $z$, so it is not $(c, d)$-tree-like for $x$ for any $c$ and $d$.\footnote{Theorem\textsuperscript{10} also describes the expressiveness (i.e., the kind of possible predictions) of $T_M$. For example, no MGNN can capture rule $r = R(x, z) \leftarrow R(x, y) \land R(y, z)$. To see this is the case, let $D = \{R(a, b), R(b, c)\}$. For an arbitrary MGNN $M$, Theorem\textsuperscript{10} ensures that all predictions of $T_M$ involving binary facts are made by rules of the form $R(x, y) \leftarrow C_x \land B_{x,y} \land C_y$; however, no such rule can derive $R(a, c)$ on $D$, so we necessarily have $R(a, c) \notin T_M(D)$ as well.}

**Theorem 10.** For each $(\text{Col}, \delta)$-MGNN $M$ and each dataset $D$, we have $T_M(D) = T_{P_M}(D)$.\footnote{Theorem\textsuperscript{10} also describes the expressiveness (i.e., the kind of possible predictions) of $T_M$. For example, no MGNN can capture rule $r = R(x, z) \leftarrow R(x, y) \land R(y, z)$. To see this is the case, let $D = \{R(a, b), R(b, c)\}$. For an arbitrary MGNN $M$, Theorem\textsuperscript{10} ensures that all predictions of $T_M$ involving binary facts are made by rules of the form $R(x, y) \leftarrow C_x \land B_{x,y} \land C_y$; however, no such rule can derive $R(a, c)$ on $D$, so we necessarily have $R(a, c) \notin T_M(D)$ as well.}

To prove Theorem\textsuperscript{10} we present a construction that, for each $(\text{Col}, \delta)$-MGNN $M$, dataset $D$, and fact $\alpha \in T_M(D)$, constructs a rule $r \in P_M$ such that $\alpha \in T_r(D)$—that is, rule $r$ explains symbolically how $M$ derives $\alpha$ on $D$. Whether such an explanation is minimal is an open question.

4 Evaluation

We evaluated our approach on KG completion—the problem of extending a KG seen as a dataset $D$ to its complete version $D'$ by adding missing facts. When seen as a classification problem, the aim is to learn a Boolean function $f(\cdot, \cdot)$ that takes a dataset $D$ and a fact $\alpha$ 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 inductive variant of this problem, 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\textsuperscript{[Bordes et al. 2013]}, where all constants in the testing data also occur in the training data.

**Baselines.** We compared our approach with DRUM\textsuperscript{[Sadeghian et al. 2019]} and AnyBURL\textsuperscript{[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 $\alpha$, returns a score representing the likelihood of $\alpha$ belonging to the completion of $D$; predictions are then computed by applying a threshold to the score. This is achieved by learning chain rules of the form $H(x_1, x_n) \leftarrow B_1(x_0, x_1) \land \cdots \land B_n(x_{n-1}, x_n)$, each associated with a confidence score.

AnyBURL uses path sampling techniques to produce a more general variant of chain rules in which variables can be replaced by constants. Each rule is associated with a score. To compute a prediction value for a dataset $D$ and fact $\alpha$, AnyBURL applies all rules to $D$ once, combines the confidence scores of all rules that derive $\alpha$, and applies a threshold to the result.

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

We evaluated our systems as follows. Each benchmark provides a method for randomly splitting its testing dataset $S$ into the incomplete dataset $S_T$ and the set $S_M$ of missing facts that should be added to $S_T$. We took $(S_T, \alpha)$ for each fact $\alpha \in S_M$ as a positive example for testing. Moreover, we randomly sampled $[S_M]$ facts $\beta_1, \ldots, \beta_n$ that use the predicates of $S$ but are not contained in $S_M$, and we took each $(S_T, \beta_j)$ as a negative example for testing. Sampling was necessary because the set of possible facts not contained in $S$ is very large; moreover, random sampling of negative examples
Table 1: Number of facts for each benchmark and phase (Training, Validation, and Testing)

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Training</th>
<th>Validation</th>
<th>Testing</th>
</tr>
</thead>
<tbody>
<tr>
<td>FB15K-237</td>
<td>4,245</td>
<td>1,838</td>
<td>7,940</td>
</tr>
<tr>
<td>NELL-995</td>
<td>489</td>
<td>856</td>
<td>1,638</td>
</tr>
<tr>
<td>WN18RR</td>
<td>2,198</td>
<td>1,000</td>
<td>1,000</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Precision</th>
<th>Recall</th>
<th>Accuracy</th>
<th>F1 Score</th>
<th>AUC</th>
<th>Training (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>FB15K-237</td>
<td>100.0</td>
<td>100.0</td>
<td>100.0</td>
<td>100.0</td>
<td>100.0</td>
<td>100.0</td>
</tr>
<tr>
<td>NELL-995</td>
<td>97.5</td>
<td>77.0</td>
<td>96.0</td>
<td>88.5</td>
<td>87.4</td>
<td>95.2</td>
</tr>
<tr>
<td>WN18RR</td>
<td>99.1</td>
<td>90.0</td>
<td>99.7</td>
<td>83.4</td>
<td>78.5</td>
<td>97.6</td>
</tr>
</tbody>
</table>

Training. Our system is trained as a denoising autoencoder [Vincent et al., 2010]. The training dataset $T$ was split with a 9:1 ratio into an incomplete dataset $T_I$ and a set $T_M$ of missing facts that should be added to $T_I$. We used $(T_I, \alpha)$ for each fact $\alpha \in T_M$ as a positive example for training, and we obtained negative examples $(T_I, \beta)$ by sampling $\beta$ analogously to how this was done for testing. We trained MGNNs with two layers and the ReLU activation function; we used cross-entropy loss with a logistic sigmoid on $\lambda^2(v)$ as the output probability; finally, we used the Adam optimisation algorithm with the standard learning rate (0.01) and weight decay ($5 \times 10^{-4}$), and a maximum of 50,000 epochs. 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 bases and their default configurations. For each system and benchmark, we computed the accuracy on the validation dataset for a range of thresholds; in our tests, we used the threshold that maximised accuracy.

Evaluation results. Table 2 shows the results of our evaluation. As one can see, our system was 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; we cannot show any times for AnyBURL since 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 is because the testing dataset $S_M$ was obtained from $S$ by random splitting so it contains many facts of the form $R(a, b)$ where constants $a$ and $b$ do not occur together in any fact in $S_I$. 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 testing dataset $S$ so that a fact $R(a, b)$ is placed into $S_M$ only if $S_I$ contains a fact of the form $S(a, b)$ or $S(b, a)$ for some predicate $S$. The results for our system obtained in this way are shown in Table 3. As one can see, precision scores remain high, whereas recall values increase significantly; thus, accuracy, F1 score, and AUC values improve accordingly. Our system significantly outperformed all baselines in this setting; however,
Table 3: Evaluation of the MGNN transformation with alternative splitting of the testing dataset

<table>
<thead>
<tr>
<th>Benchmark</th>
<th># Rules with...</th>
<th>1 body atom</th>
<th>2 body atoms</th>
</tr>
</thead>
<tbody>
<tr>
<td>FB15K-237</td>
<td>v1</td>
<td>2,102</td>
<td>1,034</td>
</tr>
<tr>
<td></td>
<td>v2</td>
<td>3,770</td>
<td>27,900</td>
</tr>
<tr>
<td></td>
<td>v3</td>
<td>3,199</td>
<td>42,590</td>
</tr>
<tr>
<td></td>
<td>v4</td>
<td>8,464</td>
<td>82,003</td>
</tr>
<tr>
<td>NELL-995</td>
<td>v1</td>
<td>149</td>
<td>215</td>
</tr>
<tr>
<td></td>
<td>v2</td>
<td>1,784</td>
<td>3,067</td>
</tr>
<tr>
<td></td>
<td>v3</td>
<td>2,306</td>
<td>5,085</td>
</tr>
<tr>
<td></td>
<td>v4</td>
<td>1,308</td>
<td>4,348</td>
</tr>
<tr>
<td>WN18RR</td>
<td>v1</td>
<td>19</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>v2</td>
<td>22</td>
<td>35</td>
</tr>
<tr>
<td></td>
<td>v3</td>
<td>23</td>
<td>19</td>
</tr>
<tr>
<td></td>
<td>v4</td>
<td>20</td>
<td>54</td>
</tr>
</tbody>
</table>

Table 4: Rules of $\mathcal{P}_M$ extracted

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Total</th>
<th>Tree-like</th>
<th>Captured by $T_M$</th>
<th>Captured on $S_I$</th>
</tr>
</thead>
<tbody>
<tr>
<td>FB15K-237</td>
<td>v1</td>
<td>2,812</td>
<td>252</td>
<td>72</td>
</tr>
<tr>
<td></td>
<td>v2</td>
<td>2,823</td>
<td>404</td>
<td>102</td>
</tr>
<tr>
<td></td>
<td>v3</td>
<td>4,805</td>
<td>490</td>
<td>118</td>
</tr>
<tr>
<td></td>
<td>v4</td>
<td>3,574</td>
<td>584</td>
<td>134</td>
</tr>
<tr>
<td>NELL-995</td>
<td>v1</td>
<td>891</td>
<td>51</td>
<td>49</td>
</tr>
<tr>
<td></td>
<td>v2</td>
<td>1,123</td>
<td>181</td>
<td>112</td>
</tr>
<tr>
<td></td>
<td>v3</td>
<td>1,104</td>
<td>277</td>
<td>181</td>
</tr>
<tr>
<td></td>
<td>v4</td>
<td>841</td>
<td>187</td>
<td>108</td>
</tr>
<tr>
<td>WN18RR</td>
<td>v1</td>
<td>397</td>
<td>15</td>
<td>11</td>
</tr>
<tr>
<td></td>
<td>v2</td>
<td>522</td>
<td>21</td>
<td>14</td>
</tr>
<tr>
<td></td>
<td>v3</td>
<td>424</td>
<td>21</td>
<td>21</td>
</tr>
<tr>
<td></td>
<td>v4</td>
<td>511</td>
<td>16</td>
<td>12</td>
</tr>
</tbody>
</table>

Table 5: Comparison of rule extraction with AnyBURL

Rule Extraction. For each benchmark, we computed using Algorithm 1 in Appendix B the subset $\mathcal{P} \subseteq \mathcal{P}_M$ of all nonredundant rules with at most two body atoms. Table 4 shows the size of each such $\mathcal{P}$. Operator $T_\mathcal{P}$ thus approximates $T_{M}$—that is, $T_\mathcal{P}(D) \subseteq T_{M}(D)$ holds for each dataset $D$. To determine the quality of the approximation, we computed the precision and recall of $T_\mathcal{P}$ on the test dataset of each benchmark; Table 5 in Appendix C shows the complete results. One can see that precision is largely the same as for $T_{M}$, while recall decreases slightly (1.6% on average). Thus, ‘short’ rules seem to account for almost all predictions on the benchmarks.

We analysed $\mathcal{P}$ manually and noticed that our transformation was able to learn numerous sound rules such as symmetry (e.g., $\text{sibling}(x,y) \leftarrow \text{sibling}(y,x)$ on FB15K-237), inverse relations (e.g., $\text{capitalOfArea}(x,y) \leftarrow \text{areaHasCapital}(y,x)$ on FB15K-237), and relationship subsumption (e.g., $\text{athletePlayedForTeam}(x,y) \leftarrow \text{athleteLedSportsTeam}(x,y)$ on NELL-995).

In addition, for each benchmark, we checked how many of the rules produced by AnyBURL are captured by our transformation; the results are shown in Table 5. For each benchmark, the first column shows the total number of rules produced by AnyBURL; the second column indicates how many of those are tree-like as per Definition 9; the third column shows how many rules from the second column are captured by our transformation as per Definition 6; and the fourth column shows the number of rules produced by AnyBURL that are captured by $T_{M}$ on the specific dataset—that is, rules $r$ that satisfy $T_{r}(S_I) \subseteq T_{M}(S_I)$. As one can see, a significant proportion of tree-like rules produced by AnyBURL are captured by $T_{M}$. Additionally, a significant proportion of (not necessarily tree-shaped) rules produced by AnyBURL are captured by $T_{M}$ on each specific dataset, which partially explains why many of the predictions by AnyBURL are also obtained by $T_{M}$.

\footnote{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 because 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 fully explained symbolically.

**Rule mining** techniques for KGs fall in two categories. Systems in the first category learn a model on training data and then extract rules from it. Neural-LP (Yang et al., 2017) is an influential such approach, and it has inspired DRUM (Sadeghian et al., 2019) and Neural-Num-LP (Wang et al., 2020). Other approaches 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 learned model and the extracted rules is unspecified in these approaches; hence, the model and the rules can make different predictions. Systems in the second category, such as AnyBURL, use training data to identify path patterns in graphs 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 train 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). ILP techniques cannot typically handle noisy examples (Muggleton, 1991; Cropper & Muggleton, 2014; Bi et al., 2019; Raghothaman et al., 2020). To improve resilience to noise, recent ILP systems such as ∂ILP (Evans & Grefenstette, 2018) interpret the ILP task as a binary classification problem and provide a differentiable implementation of deduction. These approaches compare to our work analogously to rule mining techniques. Moreover, they focus on learning rules from small datasets and generally struggle with large-scale KGs.

**Neuro-symbolic computation** aims to integrate logic with neural networks (d’Avila Garcez et al., 2002). To this end, it was shown that the immediate consequence operator of certain classes of logic programs can be approximated by recurrent (Hölldobler et al., 1999), fibering (Bader et al., 2005), and feed-forward networks (Bader et al., 2007); however, these techniques do not show how to extract a program from a trained network. A method to extract rules captured by feed-forward networks was proposed by (d’Avila Garcez et al., 2001); however, in contrast to our work, the extracted programs do not contain variables, and they are not guaranteed to be equivalent to the network. Recently, Dong et al. (2019) presented a network architecture that simulates application of function-free first-order rules to a dataset, but no algorithm for extracting rules from a trained model was provided.

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 that can provide a logical proof of each prediction.

GNNs can express 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.

Similarly to our work, techniques for neural learning of monotonic functions on the ordering of real numbers (You et al., 2017; Gupta et al., 2016; Wehenkel & Louppe, 2019) also rely on nonnegative matrices and monotonic activation functions. We, however, focus on learning monotonic functions on the ordering induced by the existence of homomorphisms between datasets.

6 CONCLUSION AND FUTURE WORK

In this paper, 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.
ACKNOWLEDGMENTS

This work was supported by the AIDA project (Alan Turing Institute, EP/N510129/1), the SIR-IUS Centre for Scalable Data Access (Research Council of Norway, project number 237889), Samsung Research UK, Siemens AG, and the EPSRC projects AnaLOG (EP/P025943/1), OASIS (EP/S032347/1) and UK FIRES (EP/S019111/1).

REFERENCES


Rianne van den Berg, Thomas N. Kipf, and Max Welling. Graph Convolutional Matrix Completion, 2017.


Peifeng Wang, Jialong Han, Chenliang Li, and Rong Pan. Logic Attention Based Neighborhood Aggregation for Inductive Knowledge Graph Embedding. In *Proc. of the 33rd AAAI Conf. on Artificial Intelligence (AAAI 2019)*, pp. 7152–7159, 2019b.


A PROOFS

A.1 PROOF OF PROPOSITION 7

Proposition 7. Let \( T \) be an operator on datasets, let \( r \) be a constant-free rule of form \( \{B_1 \mu, \ldots, B_m \mu\} \), let \( \mu \) 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 \( \mu \) that satisfy the prerequisites of the proposition. Moreover, consider an arbitrary dataset \( D \) and an arbitrary fact \( \alpha \in T(D) \); we show that \( \alpha \in T(D) \). By the definition of operator \( T \), there exists a substitution \( \nu \) 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 \( x \) occurring in \( r \). This definition of \( h \) is correct since \( \mu \) maps each variable in \( r \) to a distinct constant. Also, rule \( r \) is constant-free, so \( h(D_r) = h(\{B_1 \nu, \ldots, B_m \nu\}) = \{B_1 \mu, \ldots, B_m \mu\} \subseteq D \) holds; moreover, each constant in \( D_r \) is obtained by instantiating a variable of \( r \), so \( h \) is defined on all of \( \text{cn}(D_r) \). 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.

A.2 PROOF OF THEOREM 8

We prove Theorem 8 in two steps. First, we first introduce in Definition A1 a notion of homomorphisms on \((\text{Col}, \delta)\)-graphs that is analogous to the notion of homomorphisms on datasets. Then, in Lemma A2 we show that the transformation of \((\text{Col}, \delta)\)-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 \( a_1 \leq a_2 \) holds whenever \( (a_1)_i \leq (a_2)_i \) holds for each position \( i \).

Definition A1. A label-respecting homomorphism from a \((\text{Col}, \delta)\)-graph \( \langle V, \{E_c\}_{c \in \text{Col}}, \lambda \rangle \) to a \((\text{Col}, \delta)\)-graph \( \langle V', \{E'_c\}_{c \in \text{Col}}, \lambda' \rangle \) 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 \( c \in \text{Col} \) and each edge \( v \) in \( E_c \), between vertices \( v \) and \( w \), vertices \( h(v) \) and \( h(w) \) are connected by an edge in \( E'_c \).

Lemma A2. For each \((\text{Col}, \delta)\)-MGNN \( M \) and all \((\text{Col}, \delta)\)-graphs \( G \) and \( G' \), each label-respecting homomorphism \( h \) from \( G \) to \( G' \) is also a label-respecting homomorphism from \( M(G) \) to \( M(G') \).

Proof. Consider a \((\text{Col}, \delta)\)-MGNN \( M \) with components as specified in Definition 3 \((\text{Col}, \delta)\)-graphs \( G = \langle V, \{E_c\}_{c \in \text{Col}}, \lambda \rangle \) and \( G' = \langle V', \{E'_c\}_{c \in \text{Col}}, \lambda' \rangle \), and a label-respecting homomorphism \( h \) from \( G \) to \( G' \). Since \( M(G) \) and \( M(G') \) have the same edges as \( G \) and \( G' \), respectively, \( h \) satisfies the first condition of a label-preserving homomorphism from \( M(G) \) to \( M(G') \). 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 \( M(G) \) and \( M(G') \), 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 \( v \in V' \); since classification function \( \text{cls} \) is monotonically increasing, this will imply that \( \text{cls}(\lambda^\ell(v)) \leq \text{cls}(\lambda'^\ell(h(v))) \) holds for each vertex \( v \in V' \) 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 \( v \in V' \), and we show that \( \lambda^\ell(v) \leq \lambda'^\ell(h(v)) \) also holds for each \( v \in V' \). To this end, consider an arbitrary \( i \) between 1 and the dimension of \( B_i \). 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 9 and 10 for \( (A_i^\ell) \) and \( (B_i^\ell) \), the \( i \)-th rows of \( A^\ell \) and \( B^\ell \), respectively:

\[
\begin{align*}
x_1 &= (A_i^\ell)_i \lambda^{\ell-1}(v) + \sum_{c \in \text{Col}} (B_i^\ell)_c \max\{\lambda^{\ell-1}(w) \mid w \in N_c(v)\} + (b_i^\ell)_i, \\
x_2 &= (A_i^\ell)_i \lambda'^{\ell-1}(h(v)) + \sum_{c \in \text{Col}} (B_i^\ell)_c \max\{\lambda'^{\ell-1}(u) \mid u \in N_c(h(v))\} + (b_i^\ell)_i.
\end{align*}
\]
By the inductive hypothesis, we have \( \lambda^t(v) \leq \lambda^t(h(v)) \); moreover, all numbers in \( A^t \) are nonnegative, so we clearly have \( (A^t_i) \lambda^t(v) \leq (A^t_i) \lambda^t(h(v)) \). In addition, mapping \( h \) satisfies the first property of label-preserving homomorphisms so, for each colour \( c \in \text{Col} \) and each \( w \in N_c(v) \), we have \( h(w) \in N_c(h(v)) \); moreover, by the inductive hypothesis, we have \( \lambda^t(w) \leq \lambda^t(h(w)) \). By combining these two observations, we have

\[
\max \{ \lambda^t(w) \mid w \in N_c(v) \} \leq \max \{ \lambda^t(u) \mid u \in N_c(h(v)) \},
\]

which, together with the fact that all numbers in \( (B^t_c)_i \) are nonnegative, implies

\[
(B^t_c)_i \max \{ \lambda^t(w) \mid w \in N_c(v) \} \leq (B^t_c)_i \max \{ \lambda^t(u) \mid u \in N_c(h(v)) \}.
\]

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

**Theorem 8.** For each \((\text{Col}, \delta)\)-MGNN \( M \), mapping \( T_M \) is monotonic under homomorphisms.

**Proof.** Consider an arbitrary \((\text{Col}, \delta)\)-MGNN \( M \), 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 \((\text{Col}, \delta)\)-graphs obtained by transforming \( D \) and \( D' \) as in Definition 2. Moreover, let \( g \) be the mapping from the vertices of \( G_D \) to the vertices of \( G_D' \) as follows:

- \( g(v_a) = v_{h(a)} \) for each vertex \( v_a \) of \( G_D \); and
- \( g(v_{a,b}) = v_{h(a),h(b)} \) for each vertex \( v_{a,b} \) of \( G_D \).

It is straightforward to see that \( g \) is a label-respecting homomorphism from \( G_D \) to \( G_D' \). By Lemma A2, mapping \( g \) is also a label-respecting homomorphism from \( M(G_D) \) to \( M(G_D') \). This, however, implies that \( h \) is also a homomorphism from \( T(D) \) to \( T(D') \), as required. \( \square \)

A.3 Proof of Theorem 10

**Theorem 10.** For each \((\text{Col}, \delta)\)-MGNN \( M \) and each dataset \( D \), we have \( T_M(D) = T_{P_M}(D) \).

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

Let \( G_D \) be the \((\text{Col}, \delta)\)-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 \( v^\ell \) be the vector labelling each \( v \in V_D \) in the computation of \( M(G_D) \). Moreover, let \( \mu \) be a substitution that maps each variable to a distinct constant. We next construct an atom \( H \), a conjunction \( C \), a substitution \( \nu \) from the variables in \( C \) to \( \text{cn}(D) \), 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 \( \mu \), and mappings \( \xi : U \to V_D \) and \( M_c : U \to 2^{V_D} \) for each \( c \in \text{Col} \). 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 \( \mu \), possibly indexed letters \( v \), \( w \), and \( s \) for the vertices in \( V_D \), and possibly indexed letters \( t \) 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 \( \xi(u_{d,e}) = v_{a,b} \);

P3. conjunction \( C \) satisfies \( \nu(C) \subseteq D \); and

P4. if \( H \) is of the form \( A(x) \), conjunction \( C \) is \((3k, L)\)-tree-like for \( x \); otherwise, \( C \) is of the form \( C_x \land B_{x,y} \land C_y \), 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 \( S(y, x) \).
We initialise $C$ as the empty conjunction, and we set $\nu$, $\xi$, and each $M_c$ for $c \in \text{Col}$ as the empty mappings. For the induction base, we consider the form of the atom $\alpha \in T_M(D)$ and proceed as follows; the base case also identifies the root vertex. Properties $[\text{P1}]-[\text{P4}]$ clearly hold after these steps.

- If $\alpha$ 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 $\alpha$ 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 $u_{c,e, u_{c,d,e}}$, and $u_{c,d}$ of level $L$; we define $\xi(u_{c,e}) = v_a$, $\xi(u_{c,d}) = v_{a,b}$, and $\xi(u_{c,e,d}) = v_{a,b,a}$; and we make $u_{c,d,e}$ 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 be empty after this step. Finally, we define $M_c(u) = N_c(\xi(u))$ for each $u \in \{u_{d,e}, u_{c,d,e}\}$ and colour $c \in \text{Col}$; note that $M_{c_1}(u)$, $M_{c_2}(u)$, and $M_{c_4}(u)$ are all singletons, and $M_{c_1}(u) = \emptyset$.

Next, we consider each level $\ell$ with $1 \leq \ell \leq L$ that has already been processed, and we consider each vertex of the form $u_d$ of level $\ell$. Let $x$ be the variable such that $\mu(x) = d$, and let $a = \nu(x)$; property $[\text{P1}]$ 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

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

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 $(w^\ell)_i$, for each $\ell'$, so set $M_c(u_d)$ contains at most $\ell k$ 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 $b \in \text{cn}(D)$, depending on the set from which $v$ is drawn. Set $V_D$ clearly also contains vertices $v_{a,b}$ and $v_{b,a}$ 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 $M_{c_4}(u_d) \subseteq N_c(v_a)$ 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 $u_{c,e}$, $u_{c,d,e}$, and $u_{c,d}$ of level $\ell - 1$; and we define $\xi(u_{c,e}) = v_b$, $u_{c,d,e} = v_{a,b,a}$, and $\xi(u_{c,d}) = v_{b,a}$. 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 $u \in \{u_{d,e}, u_{c,d,e}\}$ and colour $c \in \text{Col}$; note that $M_{c_1}(u)$, $M_{c_2}(u)$, and $M_{c_4}(u)$ 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 $\ell k$ vertices and we consider three colours, this step introduces at most $3k\ell$ 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 $[\text{P1}]-[\text{P4}]$ hold after level $\ell$ 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 $C = C_x \land B_{x,y} \land C_y$ 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 $R(x, y) \leftarrow C_x \land B_{x,y} \land C_y$. To complete the proof, we next show that this rule is captured by $T_M$. Then, Definition [2] ensures $H \leftarrow C \in \mathcal{P}_M$; together with $H \nu = \alpha$ and $\nu(C) \subseteq D$, we have $\alpha \in T_{\mathcal{P}_M}(D)$, which proves our claim.

We use Proposition [7] to show that $H \leftarrow C$ is captured by $T_M$. To this end, let $D_C = \{ \alpha \nu \mid \alpha \in C \}$, and let $G_{D_{\delta}}$ be the (Col, $\delta$)-graph constructed using vertices in $U$ obtained by applying the encoding from Definition [2] to dataset $D_C$. Finally, let $u^\delta$ be the labelling associated with each vertex $u \in U$ in the computation of $M(G_{D_{\delta}})$. The construction of $C$ clearly ensures the following two properties for each colour $c \in \text{Col}$ and each vertex $u \in U$:
Algorithm 1 Rule extraction algorithm

| Input: | \( m \) : natural number (max. num. of atoms in the bodies of extracted rules) |
|        | \( \mathcal{M} \) : an MGNN |
|        | head_to_bodies := \emptyset |
| 1:     | \( m \) |
| 2:     | for \( i \leq m \) do |
| 3:     | for pred_list \in COMBINATIONS(i) do |
| 4:     | for body \in BODIES(pred_list) do |
| 5:     | if TREELIKE(body) then |
| 6:     | input_graph := ENCODE(body) |
| 7:     | output_graph := \( \mathcal{M}(\text{input_graph}) \) |
| 8:     | heads := DECODE(output_graph) |
| 9:     | for head \in heads do |
| 10:    | new_head, new_body := NORMALISE(head, body) |
| 11:    | not_implied := true |
| 12:    | for body' \in head_to_bodies[new_head] do |
| 13:    | if IMPLIES(body', new_body) then not_implied := false |
| 14:    | if not_implied then ADD(head_to_bodies[new_head], new_body) |
| 15:    | extracted_rules := \emptyset |
| 16:    | for (head, bodies) \in head_to_bodies do |
| 17:    | for body \in bodies do |
| 18:    | ADD(extracted_rules, head \leftarrow body) |

return extracted_rules

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 \( \ell \) that, for each \( 0 \leq \ell \leq L \) and each vertex \( u \in U \) whose level is at least \( \ell \), we have \( u^\ell = v^\ell \) 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 \( \xi(u_d) = v_\ast \), and the construction of \( C \) ensures \( B(x) \in C \) if and only if \( B(a) \in D \); thus, \( u_0^\ell = v_0^\ell \) 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, \( u_{d,e}^\ell = v_{a,b}^\ell \) 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 \( \ell \). Let \( v = \xi(u) \). The induction assumption ensures \( u^{\ell-1} = v^{\ell-1} \). Also, for each colour \( c \in \text{Col} \), our construction ensures that each vertex \( t \in N_c(u) \) is of level at least \( \ell - 1 \), so the induction assumption ensures \( t^{\ell-1} = w^{\ell-1} \) for \( w = \xi(t) \). Finally, property [R2] ensures that each \( t^{\ell-1} \) used to compute \( u^\ell \) is considered when computing \( v^\ell \), and property [R1] ensures that the maximal values of \( t^{\ell-1} \) are also present in the computation of \( u^\ell \). Thus, \( u^\ell = v^\ell \) holds.

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

B Optimised Rule Extraction Algorithm

We now describe the rule extraction algorithm that we used to obtain the results shown in Table 4 of Section 4. Our algorithm takes as input a positive integer \( m \) and an MGNN \( \mathcal{M} \) of dimension \( k \) and with \( L \) layers, and it produces a set of \((3k, L)\)-tree-like rules with at most \( m \) body atoms that are captured by \( T_M \). The algorithm ensures that no rule that is logically implied by any other individual rule is included in the output. The algorithm’s pseudocode is shown in Algorithm 1 and it uses several auxiliary functions that will be introduced shortly.
The algorithm initialises in line 1 an empty mapping `head_to_bodies` that will be used to store pairs of the form `(S(x,y), \{C_1, C_2, \ldots, C_n\})` such that, for each `C_i`, rule `S(x,y) \leftarrow C_i` is `(3k, L)\text{-tree-like},` has at most `m` body atoms, and is captured by `T_M`. In each iteration `i` of the main loop (lines 2–14), the algorithm produces rules with exactly `i` body atoms. To this end, the algorithm calls `COMBINATIONS(i)` in line 3 to produce all ordered combinations (with repetition) of predicates of length `i`. For each list of predicates `pred_list`, the algorithm calls `BODIES(pred_list)` in line 4 to enumerate every possibly body over the predicates of `pred_list`. For example, if `pred_list` is `[A, R]`, where `A` is unary and `R` is binary, then `BODIES(pred_list)` returns
\[
\{ A(x) \land R(x,x), A(x) \land R(x,y), A(x) \land R(y,x), A(x) \land R(y,y) \}.
\]

Next, the algorithm filters out bodies that are not `(3k, L)\text{-tree-like} (line 5), calls `ENCODER(body)` to transform `body` into a `(Col, \delta)`-graph by replacing each variable by a distinct constant and then applying the encoding from Definition 2 (line 6), applies `M` to the resulting graph (line 7), and finally calls `DECODER(output_graph)` to reverse the encoding and obtain the set `heads` of candidate head atoms (line 8). Thus, for each `head \in heads`, operator `T_M` captures `head \leftarrow body`. For each candidate `head \in heads`, the algorithm calls `NORMALISE(head, body)` to rewrite the candidate rule `head \leftarrow body` into `new_head \leftarrow new_body` to ensure that the head of each extracted rule always uses variables `x` and `y` in this order (the body is rewritten accordingly). The algorithm then checks whether rule `new_head \leftarrow new_body` is implied by any rule previously stored in `head_to_bodies` (lines 12–13), if not, it records the new rule by adding conjunction `new_body` to the set `head_to_bodies(new_head)` (line 14). Finally, the algorithm transforms the mapping `head_to_bodies` of heads to bodies into standard rules (lines 16–18).

Algorithm 1 satisfies the following property, which ensures its correctness.

**Proposition B1.** Let `P` be the result of applying Algorithm 1 to an integer `m \in \mathbb{N}` and a `(Col, \delta)`-`MGNN` `M` of dimension `k` and `L` layers.

1. Each rule `r \in P` is `(3k, L)\text{-tree-like with at most } m \text{ body atoms, and it is captured by } T_M.`
2. For each `(3k, L)\text{-tree-like rule } r \text{ with at most } m \text{ body atoms that is captured by } T_M, \text{ there exists a rule in } P \text{ that logically implies } r.

**Proof.** The first claim is a direct consequence of Proposition 7 and the fact that Algorithm 1 only considers `(3k, L)\text{-tree-like rules with at most } m \text{ atoms in the body. For the second claim, if } r \text{ is a `(3k, L)\text{-tree-like rule with at most } m \text{ body atoms, then lines 2–8 of Algorithm 1 check whether this rule is captured by } T_M; \text{ if that is the case, then lines 9–14 ensure that } r \text{ is either in the output of the algorithm, or the algorithm’s output contains a rule that logically implies } r. \Box$

## C Full Results for the Rule Extraction Experiment

As we explained in Section 4, for each benchmark, we used Algorithm 1 with `m = 2` to extract the subset `\mathcal{P} \subseteq \mathcal{P}_M` of all nonredundant rules with at most two body atoms, and then we applied `\mathcal{P}` to the testing dataset `\mathcal{S}_T` and computed the usual classification metrics. Complete results of this experiment are shown in Table 6.
Table 6: Classification metrics for the programs extracted using Algorithm 1 with $m = 2$