# On the Correspondence Between Monotonic Max-Sum GNNs and Datalog

David Tena Cucala<sup>1</sup>, Bernardo Cuenca Grau<sup>1</sup>, Boris Motik<sup>1</sup>, Egor V. Kostylev<sup>2</sup>

<sup>1</sup> Department of Computer Science, University of Oxford, UK <sup>2</sup> Department of Informatics, University of Oslo, Norway

{david.tena.cucala, bernardo.cuenca.grau, boris.motik}@cs.ox.ac.uk, egork@uio.no

#### Abstract

Although there has been significant interest in applying machine learning techniques to structured data, the expressivity (i.e., a description of what can be learned) of such techniques is still poorly understood. In this paper, we study data transformations based on graph neural networks (GNNs). First, we note that the choice of how a dataset is encoded into a numeric form processable by a GNN can obscure the characterisation of a model's expressivity, and we argue that a canonical encoding provides an appropriate basis. Second, we study the expressivity of monotonic max-sum GNNs, which cover a subclass of GNNs with max and sum aggregation functions. We show that, for each such GNN, one can compute a Datalog program such that applying the GNN to any dataset produces the same facts as a single round of application of the program's rules to the dataset. Monotonic max-sum GNNs can sum an unbounded number of feature vectors which can result in arbitrarily large feature values, whereas rule application requires only a bounded number of constants. Hence, our result shows that the unbounded summation of monotonic max-sum GNNs does not increase their expressive power. Third, we sharpen our result to the subclass of monotonic max GNNs, which use only the max aggregation function, and identify a corresponding class of Datalog programs.

### 1 Introduction

Data management tasks such as query answering or logical reasoning can be abstractly seen as transforming an input dataset into an output dataset. A key aspect of such transformations is their expressivity, which is often established by identifying a logic-based language that realises the same class of transformations. For example, core aspects of the SQL and SPARQL query languages have been characterised using fragments of first-order logic (Abiteboul, Hull, and Vianu 1995; Pérez, Arenas, and Gutierrez 2009), and logical deduction over RDF datasets has been described using the rule-based language Datalog (Motik et al. 2012). Such correspondences enable rigorous understanding and comparison of different data management languages.

Recently, there has been an increasing interest in applying machine learning techniques to data management tasks. A key benefit is that the desired transformation between datasets can be induced from examples, rather than specified explicitly. Many models have been proposed for this purpose, such as recurrent (Hölldobler, Kalinke, and Störr

1999), fibring (Bader, d'Avila Garcez, and Hitzler 2005), and feed-forward networks (Bader et al. 2007), architectures that simulate forward (Dong et al. 2019; Campero et al. 2018) and backward chaining (Rocktäschel and Riedel 2017), and architectures for rule learning (Yang, Yang, and Cohen 2017; Sadeghian et al. 2019). Graph neural networks (GNNs) have proved particularly popular since they can express graph transformations and have been widely applied to link prediction and node classification tasks in structured datasets (Schlichtkrull et al. 2018; Pflueger, Tena Cucala, and Kostylev 2022; Liu et al. 2021; Ioannidis, Margues, and Giannakis 2019; Qu, Bengio, and Tang 2019; Yang, Cohen, and Salakhutdinov 2016; Kipf and Welling 2017; Zhang and Chen 2018; Teru, Denis, and Hamilton 2020).

Characterising the expressivity of ML models for data management has thus steadily gained importance, and computational logic provides a well-established methodology: we can describe conditions under which ML-induced models become equivalent to logical formalisms in the sense that applying the ML model to an arbitrary dataset produces the same result as applying a specific logical formula. In a pioneering study, Barceló et al. (2020) showed that each GNN-induced transformation expressible in first-order logic is equivalent to a concept query of the ALCQ description logic (Baader et al. 2007)-a popular KR formalism. Huang et al. (2023) proved an analogous result for a class of GNNs with a dedicated vertex and colour. Morris et al. (2019) showed that GNNs can express certain types of graph isomorphism tests. Sourek, Zelezný, and Kuzelka (2021) characterised the expressivity of GNNs using a hybrid language where each Datalog rule is annotated with a tensor. Tena Cucala et al. (2022) characterised the expressivity of monotonic GNNs (MGNNs), which use the max aggregation function and require all weights in the matrices to be nonnegative, in terms of a class of Datalog programs. Finally, Tena Cucala, Cuenca Grau, and Motik (2022) characterised the expressivity of the Neural-LP model of rule learning.

In this paper, we take a next step in the study of the expressivity of GNN-based transformations of structured data. A key technical challenge can be summarised as follows. GNNs typically use summation to aggregate feature vectors of all vertices adjacent to a given vertex in the input graph. The number of adjacent vertices in the input is unbounded (i.e., there is no a priori limit on the number of neighbours a vertex can have), and so the summation result can be unbounded as well; hence, it appears that arbitrarily many vertices can influence whether a fact is derived. This seems fundamentally different to reasoning in fragments of first-order logic such as Datalog: the number of constants that need to be jointly considered in an application of a Datalog rule is determined by the number of rule variables, and *not* by the structure of the input dataset. Thus, at first glance, one might expect GNNs with summation to be fundamentally different from Datalog rules. To shed light on this issue, we present several novel contributions.

In Section 3 we focus on a key obstacle: to apply a GNN to a dataset, the latter must be encoded as a graph where each vertex is assigned a numeric feature vector; but then, the expressivity of the transformation inevitably depends on the details of the encoding, which obscures the contribution of the GNN itself. To overcome this, we adopt a canonical encoding, variants of which have already been considered by Schlichtkrull et al. (2018), Barceló et al. (2020), and Pflueger, Tena Cucala, and Kostylev (2022). We define a GNN to be *equivalent* to a Datalog program if applying the GNN to any dataset while using the canonical encoding produces the same facts as applying the program's rules to the dataset once (i.e., without fixpoint iteration). Finally, we observe that noncanonical encodings by Tena Cucala et al. (2022), Morris et al. (2019), or Liu et al. (2021) can be described using well-known extensions of Datalog, and so the expressivity of transformations based on such encodings can be characterised by composing all relevant programs.

In Section 4 we present our main technical contribution. First, we introduce a class of monotonic max-sum GNNs. Similarly to the MGNNs by Tena Cucala et al. (2022), monotonic max-sum GNNs require matrix weights to be be nonnegative; however, they allow for the max or sum aggregation functions in each network layer, and they place certain restrictions on the activation and classification functions (ReLU and threshold functions are allowed). Tena Cucala et al. (2022) showed that the performance of such GNNs with just max aggregation on tasks such as knowledge graph completion is on a par with that of other recent approaches. Hence, monotonic max-sum GNNs are practically relevant, but they also allow their predictions to be explained using logical proofs. Second, we prove that each monotonic max-sum GNN is equivalent to a Datalog program of a certain shape possibly containing inequalities in rule bodies. Strictly speaking, such a program can be recursive in the sense that the same predicate can occur in both rule bodies and heads; however, our notion of equivalence does not involve fixpoint iteration (i.e., the program's rules are applied just once). Thus, monotonic max-sum GNNs can derive facts with predicates from the input, but they cannot express true recursive properties such as reachability; moreover, the ability to produce unbounded feature values does not lead to a fundamental increase in expressivity. Our equivalence proof is quite different from the analogous result for MGNNs: when aggregation is limited to just max, the value of each feature of a vertex clearly depends on only a fixed number of neighbours of the vertex. Third, we prove that the equivalent Datalog program can be computed from

the GNN itself. This result is interesting because it requires enumerating potentially infinite sets of real-valued candidate feature values in a way that guarantees termination. This provides a starting point for future development of practical techniques for extracting Datalog programs from monotonic max-sum GNNs.

Finally, in Section 5 we sharpen our results to *monotonic max* GNNs, which allow only for max aggregation. We show that, analogously to MGNNs, each monotonic max GNN is equivalent to a positive Datalog program; however, we also present a converse result: we identify a class Datalog programs such that, for each program in the class, there exists an equivalent monotonic max GNN. In this way, we obtain an exact characterisation of an interesting class of GNN-based transformations using logical formalisms.

The proofs of all theorems are given in full in the extended version of this paper (Tena Cucala et al. 2023).

# 2 Preliminaries

We next recapitulate the basics of Datalog and GNNs.

**Datasets and Datalog.** We fix a signature consisting of countably infinite, disjoint sets of *predicates* and *constants*. Each predicate is associated with a nonnegative integer arity. We also consider a countably infinite set of *variables* that is disjoint with the sets of predicates and constants.

A term is a variable or a constant. An *atom* is of the form  $P(t_1, \ldots, t_n)$  where P is a predicate of arity n and  $t_1, \cdots, t_n$  are terms. An *inequality* is an expression of the form  $t_1 \not\approx t_2$  where  $t_1$  and  $t_2$  are terms. A *literal* is an atom or an inequality. A term or a literal is ground if it is variable-free. A *fact* is a ground atom and a *dataset* is a finite set of facts; thus, datasets cannot contain inequalities. A conjunction  $\alpha$  of facts is true in a dataset D, written  $D \models \alpha$ , if  $A \in D$  for each fact A in  $\alpha$ . A ground inequality  $s \not\approx t$  is true if  $s \neq t$ ; for uniformity with facts, we often write  $D \models s \not\approx t$  even though the truth of  $s \not\approx t$  does not depend on D. A (Datalog) *rule* is of the form (1) where  $n \ge 0$ ,  $B_1, \ldots, B_n$  are body literals, and H is the *head* atom:

$$B_1 \wedge \dots \wedge B_n \to H.$$
 (1)

A (Datalog) program is a finite set of rules. A substitution  $\nu$  is a mapping of finitely many variables to ground terms; for  $\alpha$  a literal,  $\alpha\nu$  is the result of replacing in  $\alpha$  each variable x with  $\nu(x)$  provided the latter is defined. Each rule r of form (1) defines an *immediate consequence* operator  $T_r$  on datasets: for D a dataset,  $T_r(D)$  is the dataset that contains the fact  $H\nu$  for each substitution  $\nu$  mapping all variables of r to terms occurring in D such that  $D \models B_i\nu$  for each  $1 \le i \le n$ . For  $\mathcal{P}$  a program,  $T_{\mathcal{P}}(D) = \bigcup_{r \in \mathcal{P}} T_r(D)$ .

To simplify the formal treatment, we do not make the usual *safety* requirement where each variable in a rule must occur in a body atom; in fact, the body can be empty, which we denote by  $\top$ . For example, rule  $r = \top \rightarrow R(x, y)$  is syntactically valid; moreover, the definition of  $T_r$  ensures that  $T_r(D)$  contains exactly each fact R(s, t) for all (not necessarily distinct) terms s and t occurring in D.

Conjunctions  $\alpha$  and  $\beta$  of literals are *equal up to variable* renaming if there exists a bijective mapping  $\nu$  from the set of all variables of  $\alpha$  to the set of all variables of  $\beta$  such that  $\alpha\nu$  and  $\beta$  contain exactly the same conjuncts; this notion is extended to rules in the obvious way. A set *S* contains a conjunction  $\alpha$  of literals *up to variable renaming* if there exists  $\beta \in S$  such that  $\alpha$  and  $\beta$  are equal up to variable renaming.

**Graph Neural Networks.** We use  $\mathbb{R}$  and  $\mathbb{R}_0^+$  for the sets of real and nonnegative real numbers, respectively. Also, we use  $\mathbb{N}$  for the set of natural numbers, and  $\mathbb{N}_0 = \mathbb{N} \cup \{0\}$ .

A function  $\sigma : \mathbb{R} \to \mathbb{R}$  is *monotonically increasing* if x < y implies  $\sigma(x) \leq \sigma(y)$ . Function  $\sigma$  is *Boolean* if its range is  $\{0, 1\}$ . Finally,  $\sigma$  is *unbounded* if, for each  $y \in \mathbb{R}$ , there exists  $x \in \mathbb{R}$  such that  $\sigma(x) > y$ .

A real *multiset* is a function  $S : \mathbb{R} \to \mathbb{N}_0$  that assigns to each  $x \in \mathbb{R}$  the number of occurrences S(x). Such S is *finite* if S(x) > 0 for finitely many  $x \in \mathbb{R}$ ; the *cardinality* of such S is  $|S| = \sum_{x \in \mathbb{R}} S(x)$ ; and  $\mathcal{F}(\mathbb{R})$  is the set of all finite real multisets. We often write a finite S as a list of possibly repeated real numbers in double-braces  $\{\!\{\dots\}\!\}$ . Finally, we treat a set as a multiset where each element occurs just once.

We consider vectors and matrices over  $\mathbb{R}$  and  $\mathbb{R}_0^+$ . For  $\mathbf{v}$  a vector and i a natural number,  $(\mathbf{v})_i$  is the *i*-th element of  $\mathbf{v}$ . We apply scalar functions to vectors element-wise; for example, given n vectors  $\mathbf{v}_1, \ldots, \mathbf{v}_n$  of equal dimension,  $\max{\{\mathbf{v}_1, \ldots, \mathbf{v}_n\}}$  is the vector whose *i*-th element is equal to  $\max{\{(\mathbf{v}_1)_i, \ldots, (\mathbf{v}_n)_i\}}$ .

For Col a finite set of *colours* and  $\delta \in \mathbb{N}$  a *dimension*, a (Col,  $\delta$ )-graph is a tuple  $\mathcal{G} = \langle \mathcal{V}, \{\mathcal{E}^c\}_{c \in Col}, \lambda \rangle$  where  $\mathcal{V}$ is a finite set of *vertices*; for each  $c \in Col$ ,  $\mathcal{E}^c \subseteq \mathcal{V} \times \mathcal{V}$ is a set of directed *edges*; and *labelling*  $\lambda$  assigns to each  $v \in \mathcal{V}$  a *feature* vector  $\mathbf{v}_{\lambda}$  of dimension  $\delta$ . Graph  $\mathcal{G}$  is *symmetric* if  $\langle v, u \rangle \in \mathcal{E}^c$  implies  $\langle u, v \rangle \in \mathcal{E}^c$  for each  $c \in Col$ , and it is *Boolean* if  $(\mathbf{v}_{\lambda})_i \in \{0, 1\}$  for each  $v \in \mathcal{V}$  and  $i \in \{1, \ldots, \delta\}$ . To improve readability, we abbreviate  $\mathbf{v}_{\lambda}$  to just  $\mathbf{v}$  when the labelling function is clear from the context; analogously, we abbreviate  $\mathbf{v}_{\lambda_{\ell}}$  to  $\mathbf{v}_{\ell}$ .

A (Col,  $\delta$ )-graph neural network (GNN)  $\mathcal{N}$  with  $L \geq 1$  layers is a tuple

$$\begin{array}{l} \langle \{\mathbf{A}_{\ell}\}_{1 \leq \ell \leq L}, \{\mathbf{B}_{\ell}^{c}\}_{c \in \mathsf{Col and } 1 \leq \ell \leq L}, \\ \{\mathbf{b}_{\ell}\}_{1 \leq \ell \leq L}, \{\mathsf{agg}_{\ell}\}_{1 \leq \ell \leq L}, \sigma, \mathsf{cls} \rangle, \end{array}$$
(2)

where, for each  $\ell \in \{1, \ldots, L\}$  and  $c \in \text{Col}$ ,  $\mathbf{A}_{\ell}$  and  $\mathbf{B}_{\ell}^{c}$  are matrices over  $\mathbb{R}$  of dimension  $\delta_{\ell} \times \delta_{\ell-1}$  with  $\delta_{0} = \delta_{L} = \delta$ ,  $\mathbf{b}_{\ell}$  is a vector over  $\mathbb{R}$  of dimension  $\delta_{\ell}$ ,  $\operatorname{agg}_{\ell} : \mathcal{F}(\mathbb{R}) \to \mathbb{R}$  is an *aggregation* function,  $\sigma : \mathbb{R} \to \mathbb{R}$  is an *activation* function, and cls :  $\mathbb{R} \to \{0, 1\}$  is a *classification* function.

Applying  $(Col, \delta)$ -GNN  $\mathcal{N}$  to  $(Col, \delta)$ -graph  $\mathcal{G}$  induces the sequence  $\lambda_0, \ldots, \lambda_L$  of vertex labelling functions such that  $\lambda_0 = \lambda$  and, for each  $\ell \in \{1, \ldots, L\}$  and  $v \in V$ , the value of  $\mathbf{v}_{\ell}$  is given by

$$\mathbf{v}_{\ell} = \sigma \Big( \mathbf{A}_{\ell} \mathbf{v}_{\ell-1} + \sum_{c \in \mathsf{Col}} \mathbf{B}_{\ell}^{c} \operatorname{agg}_{\ell} \big( \{\!\!\{ \mathbf{u}_{\ell-1} \mid \langle v, u \rangle \in \mathcal{E}^{c} \}\!\!\} \big) + \mathbf{b}_{\ell} \Big).$$
<sup>(3)</sup>

The result  $\mathcal{N}(\mathcal{G})$  of applying  $\mathcal{N}$  to  $\mathcal{G}$  is the Boolean (Col,  $\delta$ )graph with the same vertices and edges as  $\mathcal{G}$ , but where each vertex  $v \in \mathcal{V}$  is labelled by  $cls(\mathbf{v}_L)$ .

# **3** Choosing an Encoding/Decoding Scheme

To realise a dataset transformation using a GNN, we must first encode the input dataset into a graph that can be processed by a GNN, and subsequently decode the GNN's output back into a dataset. Several encoding/decoding schemes have been proposed in the literature, and their details differ considerably. As a result, when characterising GNN-based transformations of datasets using logic, it can be hard to understand which properties of the characterisation are due to the chosen encoding/decoding scheme, and which are immanent to the GNN used to realise the transformation. In this paper we consider primarily the encoding scheme that straightforwardly converts a dataset into a graph, but we also discuss how to take other encoding schemes into account.

#### 3.1 Canonical Encoding/Decoding Scheme

A straightforward way to encode a dataset containing only unary and binary facts into a Boolean  $(Col, \delta)$ -graph is to transform terms into vertices, use vertex connectivity to describe binary facts, and encode presence of unary facts in feature vectors. Such encoding/decoding schemes, which we call *canonical*, have already been widely used in the literature with minor variations (Schlichtkrull et al. 2018; Pflueger, Tena Cucala, and Kostylev 2022; Barceló et al. 2020). They establish a direct syntactic correspondence between datasets and coloured graphs and are thus a natural starting point for studying the expressivity of GNNs.

We next describe one such scheme. In particular, we introduce  $(Col, \delta)$ -datasets, which naturally correspond to a large class of  $(Col, \delta)$ -graphs. Our definitions provide the foundation necessary to formulate our expressivity results in Section 4. In Section 3.2 we discuss how to combine our expressivity results with more complex encoding schemes.

**Definition 1.** Let Col be a set of colours and let  $\delta \in \mathbb{N}$  be a dimension. A (Col,  $\delta$ )-signature contains

- a binary predicate  $E^c$  for each colour  $c \in Col$ , and
- a unary predicate  $U_i$  for each  $i \in \{1, \ldots, \delta\}$ .

A (Col,  $\delta$ )-fact has a predicate from the (Col,  $\delta$ )-signature, and a (Col,  $\delta$ )-dataset contains only (Col,  $\delta$ )-facts.

We assume that terms occurring in datasets correspond one-to-one to vertices of coloured graphs—that is, each term t is paired with a unique vertex  $v_t$ . This is again without loss of generality since the result of applying a GNN to a coloured graph does not depend on the identity of vertices, but only on the graph structure and the feature vectors.

We are now ready to define the canonical GNN-based transformations of  $(Col, \delta)$ -datasets.

**Definition 2.** *The* canonical encoding enc(D) of a (Col,  $\delta$ )dataset D is the Boolean (Col,  $\delta$ )-graph  $\langle \mathcal{V}, \{\mathcal{E}^c\}_{c \in Col}, \lambda \rangle$ defined as follows:

- $\mathcal{V}$  contains the vertex  $v_t$  for each term t occurring in D;
- $\langle v_t, v_s \rangle \in \mathcal{E}^c$  if  $E^c(t, s) \in D$  for each  $c \in \mathsf{Col}$ ; and
- $(\mathbf{v}_t)_i = 1$  if  $U_i(t) \in D$ , and  $(\mathbf{v}_t)_i = 0$  otherwise.

The canonical decoding dec( $\mathcal{G}$ ) of a Boolean (Col,  $\delta$ )-graph  $\mathcal{G} = \langle \mathcal{V}, \{\mathcal{E}^c\}_{c \in Col}, \lambda \rangle$  is the dataset that contains

• the fact  $E^{c}(t,s)$  for each  $\langle v_{t}, v_{s} \rangle \in \mathcal{E}^{c}$  and  $c \in \text{Col}$ , and

• the fact  $U_i(t)$  for each  $v_t \in \mathcal{V}$  and  $i \in \{1, \ldots, \delta\}$  such that  $(\mathbf{v}_t)_i = 1$ .

Each  $(Col, \delta)$ -GNN  $\mathcal{N}$  induces the canonical transformation  $T_{\mathcal{N}}$  on  $(Col, \delta)$ -datasets where  $T_{\mathcal{N}}(D) = dec(\mathcal{N}(enc(D)))$  for each  $(Col, \delta)$ -dataset D.

This encoding neither introduces nor omits any information from the input dataset, so a  $(Col, \delta)$ -dataset D and its canonical encoding enc(D) straightforwardly correspond to one another. Since datasets are directional,  $(Col, \delta)$ -graphs must be directed as well to minimise the discrepancy between the two representations. The canonical decoding is analogous to the encoding, and the two are inverse operations on graphs that are regular as per Definition 3.

**Definition 3.** A  $(Col, \delta)$ -graph  $\mathcal{G} = \langle \mathcal{V}, \{\mathcal{E}^c\}_{c \in Col}, \lambda \rangle$  is regular if  $\mathcal{G}$  is Boolean and each vertex  $v \in \mathcal{V}$  either occurs in  $\mathcal{E}^c$  for some  $c \in Col, or (\mathbf{v})_i = 1$  for some  $i \in \{1, \ldots, \delta\}$ .

Our canonical encoding produces only regular graphs, and there is a one-to-one correspondence between  $(Col, \delta)$ datasets and regular (Col,  $\delta$ )-graphs. Our results from the following sections can be equivalently framed as characterising expressivity of GNN transformations of regular graphs in terms of Datalog programs. Graphs that are not Boolean do not correspond to encodings of datasets, so we do not see a natural way to view GNN transformations over such graphs in terms of logical formalisms. Finally, a (Col,  $\delta$ )graph  $\mathcal{G}$  that is Boolean but not regular contains 'isolated' vertices that are not connected to any other vertex and are labelled by zeros only. When such  $\mathcal{G}$  is decoded into a (Col,  $\delta$ )-dataset, such 'isolated' vertices do not produce any facts in  $dec(\mathcal{G})$  and thus several non-regular Boolean graphs can produce the same (Col,  $\delta$ )-dataset. Note, however, that each 'isolated' zero-labelled vertex is transformed by a GNN in the same way-that is, the vector labelling the vertex in the GNN's output does not depend on any other vertices but only on the matrices of the GNN. Consequently, such vertices are not interesting for our study of GNN expressivity.

We are now ready to formalise our central notion of equivalence between a GNN and a Datalog program.

**Definition 4.** A (Col,  $\delta$ )-GNN  $\mathcal{N}$  captures a rule or a Datalog program  $\alpha$  if  $T_{\alpha}(D) \subseteq T_{\mathcal{N}}(D)$  for each (Col,  $\delta$ )-dataset D. Moreover,  $\mathcal{N}$  and  $\alpha$  are equivalent if  $T_{\mathcal{N}}(D) = T_{\alpha}(D)$ for each (Col,  $\delta$ )-dataset D.

The key question we address in Sections 4 and 5 is the following: under what conditions is a given (Col,  $\delta$ )-GNN  $\mathcal{N}$  equivalent to a Datalog program, and can this program (at least in principle) be computed from  $\mathcal{N}$ ?

#### 3.2 Noncanonical Encoding/Decoding Schemes

For each  $(Col, \delta)$ -dataset D, the binary facts of D and  $T_{\mathcal{N}}(D)$  coincide, and so applying  $T_{\mathcal{N}}$  to D cannot derive any binary facts. To overcome this limitation, more complex, noncanonical encodings have been proposed (Tena Cucala et al. 2022; Morris et al. 2019; Liu et al. 2021). These introduce vertices representing combinations of several constants so that facts of higher arity can be encoded in appropriate feature vectors, but there is no obvious canonical way to achieve this. Expressivity results based on such encodings

are less transparent because it is not obvious which aspects of expressivity are due to the encoding/decoding scheme and which are immanent to the GNN itself.

We argue that noncanonical encoding/decoding schemes can often be described by a pair of programs  $\mathcal{P}_{enc}$  and  $\mathcal{P}_{dec}$ , possibly expressed in a well-known extension of Datalog, which convert an input dataset into a  $(Col, \delta)$ -dataset and vice versa. Thus, given an arbitrary dataset D, the result of applying the end-to-end transformation that uses a GNN  $\mathcal{N}$  and the respective encoding/decoding scheme is  $T_{\mathcal{P}_{dec}}(T_{\mathcal{N}}(T_{\mathcal{P}_{enc}}(D)))$ . Furthermore, if  $\mathcal{N}$  is equivalent to a Datalog program  $\mathcal{P}_{\mathcal{N}}$ , then the composition of  $\mathcal{P}_{enc}$ ,  $\mathcal{P}_{\mathcal{N}}$ , and  $\mathcal{P}_{dec}$  characterises the end-to-end transformation. This allows us to clearly separate the contribution of the GNN from the contributions of the encoding and decoding.

Tena Cucala et al. (2022) recently presented a dataset transformation based on a class of monotonic GNNs (MGNNs). Their approach is applicable to a dataset D that uses unary predicates  $A_1, \ldots, A_{\epsilon}$  and binary predicates  $R_{\epsilon+1}, \ldots, R_{\delta}$ , and D is encoded into a symmetric (Col,  $\delta$ )-graph over the set of colours  $Col = \{c_1, c_2, c_3, c_4\}$ . The encoding introduces a vertex  $v_a$  for each constant a in D as well as vertices  $v_{a,b}$  and  $v_{b,a}$  for each pair of constants a, b occurring together in a binary fact in D. Predicates are assigned fixed positions in vectors so that the value of a component of a vector labelling a vertex indicates the presence or absence of a specific fact in D. For example, if  $A_i(a) \in D$ , then  $(\mathbf{v}_a)_i$ is set to 1; analogously, if  $R_j(a, b) \notin D$  but a and b occur in D in a binary fact, then  $(\mathbf{v}_{a,b})_j$  is set to 0. Moreover, the edges of the coloured graph indicate different types of 'connections' between constants; for example, vertices  $v_a$  and  $v_{a,b}$  are connected by an edge of colour  $c_1$  to indicate that constant a occurs first in the constant pair (a, b). A variant of this approach was also proposed by Liu et al. (2021) in the context of knowledge graph completion.

We next show how to capture this encoding using rules. Note that the encoder introduces vertices of the form  $v_{a,b}$  for pairs of constants a and b, so the encoding program  $\mathcal{P}_{enc}$  requires value invention. This can be conveniently realised using functional terms. For example, we can represent vertex  $v_{a,b}$  using term g(a,b), and we can represent each vertex of the form  $v_a$  using a term f(a) for uniformity. Applying the encoding program  $\mathcal{P}_{enc}$  to a dataset thus produces a (Col,  $\delta$ )-dataset with functional terms, which should be processed by the GNN as if they were constants; for example, the canonical encoding should transform g(a, b) into vertex  $v_{g(a,b)}$ . Based on this idea, the encoding program  $\mathcal{P}_{enc}$  contains rule (4) instantiated for each  $i \in \{1, \ldots, \delta\}$ , and rules (5)–(13) instantiated for each  $j \in \{\epsilon + 1, \ldots, \delta\}$ .

$$A_i(x) \to U_i(f(x)) \tag{4}$$

$$R_j(x,y) \to U_j(g(x,y)) \tag{5}$$

$$R_j(x,y) \to E^{c_1}(f(x),g(x,y)) \tag{6}$$

- $R_j(x,y) \to E^{c_1}(g(x,y), f(x))$  (7)
- $R_j(x,y) \to E^{c_2}(f(y),g(x,y)) \tag{8}$
- $R_j(x,y) \to E^{c_2}(g(x,y), f(y)) \tag{9}$

$$R_j(x,y) \to E^{c_3}(g(x,y),g(y,x))$$
 (10)

$$R_j(x,y) \to E^{c_3}(g(y,x),g(x,y))$$
 (11)

$$R_j(x,y) \to E^{c_4}(f(x), f(y)) \tag{12}$$

$$R_j(x,y) \to E^{c_4}(f(y), f(x)) \tag{13}$$

Rules (4) and (5) ensure that all unary and binary facts in the input dataset are encoded as facts of the form  $U_i(f(a))$ and  $U_j(g(a, b))$ ; thus, when these are further transformed into a (Col,  $\delta$ )-graph, the vectors labelling vertices  $v_{f(a)}$ and  $v_{g(a,b)}$  encode all input facts of the form  $A_i(a)$  and  $R_j(a, b)$  for  $i \in \{1, \ldots, \epsilon\}$  and  $j \in \{\epsilon + 1, \ldots, \delta\}$ . In addition, rules (6)–(13) encode the adjacency relationships between terms: colour  $c_1$  connects terms g(a, b) and f(a), colour  $c_2$  connects g(a, b) and f(b), colour  $c_3$  connects g(a, b) and g(b, a), and colour  $c_4$  connects terms f(a) and f(b) provided that a and b occur jointly in a binary fact.

Program  $\mathcal{P}_{dec}$  capturing the decoder contains rule (14) instantiated for each  $i \in \{1, \ldots, \epsilon\}$ , as well as rule (15) instantiated for each  $j \in \{\epsilon + 1, \ldots, \delta\}$ .

$$U_i(f(x)) \to A_i(x)$$
 (14)

$$U_j(g(x,y)) \to R_j(x,y) \tag{15}$$

Intuitively, these rules just 'read off' the facts from the labels of vertices such as  $v_{f(a)}$  and  $v_{g(a,b)}$ . The composition of these three programs is a (function-free) Datalog program.

It is straightforward to show that, for each dataset D, the graph obtained by applying the encoder by Tena Cucala et al. (2022) is isomorphic to the graph obtained by applying the canonical encoding from Definition 2 to  $T_{\mathcal{P}_{enc}}(D)$  and thus program  $\mathcal{P}_{enc}$  correctly captures their encoder.

A limitation of this encoding is that the transformation's output can contain a fact of the form R(a, b) only if the input dataset contains a fact of the form S(a, b) or S(b, a). Intuitively, the presence of S(a, b) or S(b, a) in the input ensures that the resulting (Col,  $\delta$ )-graph contains a vertex  $v_{q(a,b)}$  for representing binary facts of the form R(a, b). An obvious way to overcome this limitation is to introduce terms q(a, b)for all constants a and b occurring in the input, without requiring a and b to occur jointly in a binary fact. While this increases the expressivity of the end-to-end transformation, the increase is due to the encoding step, rather than the GNN. Our framework makes this point clear. For example, we can extend  $\mathcal{P}_{enc}$  with rules such as (16)–(19) and so on for all other combinations of unary and binary predicates and colours. The chaining of  $\mathcal{P}_{enc}$ ,  $\mathcal{P}_{\mathcal{N}}$ , and  $\mathcal{P}_{dec}$  can now capture different transformations even if  $\mathcal{P}_{\mathcal{N}}$  remains the same.

$$A_i(x) \wedge A_j(y) \to E^{c_1}(f(x), g(x, y)) \tag{16}$$

$$A_i(x) \wedge A_j(y) \to E^{c_1}(g(x,y), f(x)) \tag{17}$$

$$R_i(x,z) \wedge A_j(y) \to E^{c_1}(g(x,y),f(x)) \tag{18}$$

$$R_i(z,x) \wedge A_j(y) \to E^{c_1}(g(x,y), f(x))$$
(19)

**Morris et al. (2019)** introduced *k*-GNNs and showed them to be more expressive than standard GNNs. The input to a *k*-GNN is a symmetric (Col,  $\delta_1$ )-graph  $\mathcal{G}_1$  without self-loops where Col contains a single colour *c* and, for each vertex *v* of  $\mathcal{G}_1$ ,  $(\mathbf{v})_i = 1$  for exactly one  $1 \le i \le \delta_1$ . To apply a *k*-GNN to  $\mathcal{G}_1$ , the latter is transformed into another (Col,  $\delta_2$ )-graph  $\mathcal{G}_2$  that contains one vertex for each set of *k* distinct vertices of  $\mathcal{G}_1$ , and then a standard (Col,  $\delta_2$ )-GNN is applied to  $\mathcal{G}_2$ . We next show that the transformation of  $\mathcal{G}_1$  into  $\mathcal{G}_2$  can be captured by a program  $\mathcal{P}_{enc}$  that transforms a  $(Col, \delta_1)$ dataset over unary predicates  $A_1, \ldots, A_{\delta_1}$  and a binary predicate R into a  $(Col, \delta_2)$ -dataset. Thus, the increase in expressivity of k-GNNs does not come from the GNN model itself, but rather from the encoding implicit in their approach. For readability, we make several simplifying assumptions. First, while Morris et al. (2019) consider sets of k distinct vertices in order to ensure practical scalability, we consider k-tuples instead and limit our presentation to just k = 2. Second, we consider just the *local neighbourhood* approach to connecting vertices in  $\mathcal{G}_2$ . Finally, our encoding requires extending Datalog not only with function symbols, but also with stratified negation-as-failure not (Dantsin et al. 2001).

Program  $\mathcal{P}_{enc}$  consists of rules (20)–(23) instantiated for all  $i, j, k, \ell \in \{1, \dots, \delta_1\}$ .

$$\begin{array}{l}
A_i(x) \wedge A_j(y) \wedge x \not\approx y \wedge \\
A_k(x) \wedge A_\ell(z) \wedge x \not\approx z \wedge \\
R(y, z) \wedge y \not\approx z \rightarrow E^c(g(x, y), g(x, z))
\end{array}$$
(20)

$$\begin{array}{l}
A_i(y) \wedge A_j(x) \wedge y \not\approx x \wedge \\
A_k(z) \wedge A_\ell(x) \wedge z \not\approx x \wedge \\
R(y,z) \wedge y \not\approx z \rightarrow E^c(g(y,x),g(z,x))
\end{array}$$
(21)

$$A_i(x) \wedge A_j(y) \wedge x \not\approx y \wedge \operatorname{not} R(x, y) \to U_{i,j,0}(g(x, y))$$
(22)

$$A_{i}(x) \wedge A_{j}(y) \wedge x \not\approx y \wedge R(x, y)$$
  
 
$$\rightarrow U_{i,j,1}(g(x, y))$$
 (23)

Conjunctions of the form  $A_i(x) \wedge A_j(y) \wedge x \not\approx y$  in these rules identify pairs of distinct constants a and b (corresponding to the vertices of  $\mathcal{G}_1$ ) in the input dataset, and, for each such pair, g(x, y) introduces a term g(a, b) (corresponding to a vertex of  $\mathcal{G}_2$ ). Rules (20) and (21) encode the *local* neighbourhood approach: terms g(a, b) and g(d, e) are connected in  $\mathcal{G}_2$  if either a = b and  $d \neq e$ , or  $a \neq b$  and d = e, and additionally the two constants in the inequality are connected in  $\mathcal{G}_1$ . Finally, rules (22) and (23) identify the type of the subgraph of  $\mathcal{G}_1$  that a and b participate in. Specifically, a fact of the form  $U_{i,j,0}(g(a, b))$  says that a and b are labelled in  $\mathcal{G}_1$ . A fact of the form  $U_{i,j,1}(g(a, b))$  is analogous, but with the difference that a and b are connected in  $\mathcal{G}_1$ .

#### 4 GNNs with Max-Sum Aggregation

In this section, we introduce monotonic max-sum GNNs and prove that each such GNN corresponds to a Datalog program (possibly with inequalities in the rule bodies) that can be computed from the GNN's definition. Monotonic maxsum GNNs can use the following aggregation function in all layers, which generalises both max and sum.

**Definition 5.** For  $k \in \mathbb{N}_0 \cup \{\infty\}$ , a finite real multiset  $S \in \mathcal{F}(\mathbb{R})$ , and  $\ell = \min(k, |S|)$ , let

$$\max\text{-}k\text{-}\mathrm{sum}(S) = \begin{cases} 0 & \text{if } \ell = 0, \\ \underset{i=1}{\ell} s_i & \text{where } s_1, \dots, s_\ell \text{ are the} \\ \ell \text{ largest numbers of } S. \end{cases}$$

Each occurrence of a number is counted separately; for example,  $\max$ -3-sum( $\{\!\{0, 1, 1, 2, 2, 5\}\!\}$ ) = 9 because the

three largest numbers in S are 5 and the two occurrences of 2. Also, max-1-sum is equivalent to max, and max- $\infty$ -sum is equivalent to sum; hence,  $\max$ -sum generalises both the max and sum aggregation functions. While the ability to sum just the k maximal elements may not be relevant in practice, it will allow us to formalise a key technical result. We next introduce monotonic max-sum GNNs.

**Definition 6.** A monotonic max-sum  $(Col, \delta)$ -GNN is a GNN of form (2) satisfying the following conditions:

- for each  $\ell \in \{1, \ldots, L\}$  and each  $c \in Col$ , all elements of matrices  $\mathbf{A}_{\ell}$  and  $\mathbf{B}_{\ell}^{c}$  are nonnegative;
- for each  $\ell \in \{1, \ldots, L\}$ , the aggregation function  $agg_{\ell}$  is max- $k_{\ell}$ -sum for some  $k_{\ell} \in \mathbb{N}_0 \cup \{\infty\}$ ;
- the activation function  $\sigma$  is monotonically increasing and unbounded, and the range of  $\sigma$  is  $\mathbb{R}_0^+$ ; and
- *the classification function* cls *is a step function*—*that is,* there exists a threshold  $t \in \mathbb{R}$  such that cls(t') = 0 for each t' < t, and cls(t') = 1 for each  $t' \ge t$ .

Monotonic max-sum GNNs are closely related to, but incomparable with MGNNs by Tena Cucala et al. (2022): MGNNs do not require the activation function to be unbounded, but they support only the max aggregation function in all layers. While ReLU satisfies Definition 6, neither ELU nor the sigmoid function is compatible.

In Section 4.1, we show that, in each monotonic maxsum GNN  $\mathcal{N}$ , one can replace each function max- $k_{\ell}$ -sum where  $k_{\ell} = \infty$  with max- $C_{\ell}$ -sum for some  $C_{\ell} \in \mathbb{N}_0$  without changing the canonical transformation induced by  $\mathcal{N}$  that is, to apply a GNN to a dataset, we need to consider only a bounded number of vertices for aggregation. Number  $C_{\ell}$ depends solely on  $\mathcal{N}$  (i.e., it is independent of any dataset to which  $\mathcal{N}$  is applied) and is called the *capacity* of layer  $\ell$ . In Section 4.2, we use this result to show that  $T_N$  is equivalent to the immediate consequence operator of a Datalog program  $\mathcal{P}_{\mathcal{N}}$  that depends only on  $\mathcal{N}$ . Finally, in Section 4.3, we show that the numbers  $C_{\ell}$  can be computed from  $\mathcal{N}$ , and hence program  $\mathcal{P}_{\mathcal{N}}$  is computable. Our objective is to show that extracting  $\mathcal{P}_{\mathcal{N}}$  from  $\mathcal{N}$  is possible in principle, but further work is needed to devise a practical procedure.

### 4.1 Limiting Neighbour Aggregation

Throughout the rest of Section 4, we fix a monotonic maxsum (Col,  $\delta$ )-GNN  $\mathcal{N}$  of form (2) and dimensions  $\delta_0, \ldots, \delta_L$ as specified in Section 2, and we fix  $k_1, \ldots, k_L$  as the numbers defining the aggregation functions of  $\mathcal{N}$ . We next show that each  $k_{\ell} = \infty$  can be replaced with a natural number  $C_{\ell}$ . We first introduce several auxiliary definitions.

**Definition 7.** A (Col,  $\ell$ )-multiset family, where  $0 \le \ell \le L$ , is a mapping **Y** that assigns to each colour  $c \in Col$  a finite multiset  $\mathbf{Y}^c$  of vectors of dimension  $\delta_{\ell}$ .

For each  $1 \leq \ell \leq L$ , each  $1 \leq i \leq \delta_{\ell}$ , each vector **x** of dimension  $\delta_{\ell-1}$ , and each (Col,  $\ell-1$ )-multiset family **Y**, let

$$\mathsf{Val}(\ell, i, \mathbf{x}, \mathbf{Y}) = (\mathbf{A}_{\ell} \mathbf{x} + \sum_{c \in \mathsf{Col}} \mathbf{B}_{\ell}^{c} \max k_{\ell} \operatorname{sum}(\mathbf{Y}^{c}) + \mathbf{b}_{\ell})_{i}.$$

Sets  $\mathcal{X}_{\ell,i}$  with  $0 \leq \ell \leq L$  and  $1 \leq i \leq \delta_{\ell}$  are defined by induction on  $\ell$  as follows.

#### Algorithm 1 CAPACITY( $\mathcal{N}$ )

- 1: let  $\alpha_L$  be the threshold of cls
- 2: for  $\ell$  from L down to 1 do
- $w_{\ell} :=$  the least non-zero element of  $\mathbf{A}_{\ell}$  and all  $\mathbf{B}_{\ell}^{c}$ 3:
- 4:  $\epsilon_{\ell} :=$  the least non-zero number in  $\bigcup_{i} \mathcal{X}_{\ell-1,i}$
- 5: if either  $w_\ell$  or  $\epsilon_\ell$  does not exist then
- 6:  $C_{\ell} := C_{\ell-1} := C_1 := 0$
- 7: return
- 8:  $\beta_{\ell}$  := the least natural number such that  $\sigma(\beta_{\ell}) \geq \alpha_{\ell}$

9: 
$$b_{\ell} :=$$
 the least element of b

 $C_{\ell} := \min(k_{\ell}, \lceil \frac{\beta_{\ell} - b_{\ell}}{w_{\ell} \cdot \epsilon_{\ell}} \rceil)$  $\alpha_{\ell-1} := \frac{\beta_{\ell} - b_{\ell}}{w_{\ell}}$ 10:

11:

- For each  $1 \le i \le \delta_0$ , let  $\mathcal{X}_{0,i} = \{0, 1\}$ .
- For each  $\ell \geq 1$  and each  $1 \leq i \leq \delta_{\ell}$ , set  $\mathcal{X}_{\ell,i}$  is the least set that contains  $\sigma(Val(\ell, i, \mathbf{x}, \mathbf{Y}))$  for each vector **x** of dimension  $\delta_{\ell-1}$  such that  $(\mathbf{x})_j \in \mathcal{X}_{\ell-1,j}$  for each j, and each  $(\text{Col}, \ell - 1)$ -multiset family **Y** such that  $(\mathbf{y})_j \in \mathcal{X}_{\ell-1,j}$  for all  $c \in \mathsf{Col}, \mathbf{y} \in \mathbf{Y}^c$ , and j.

Intuitively, sets  $\mathcal{X}_{\ell,i}$  contain all real numbers that can occur in the *i*-th position of a vector labelling a vertex at layer  $\ell$ when  ${\mathcal N}$  is applied to a canonical encoding of some (Col,  $\ell)\text{-}$ dataset. Indeed, by the base case of the definition,  $\mathcal{X}_{0,i}$  contains all values that can be produced by the canonical encoding, and the inductive step considers all possible ways in which a vector in layer  $\ell$  can be computed from vectors in layer  $\ell - 1$  using propagation equation (3). In the latter case, a (Col,  $\ell$ )-multiset family **Y** represents a collection of possible neighbour vectors, and  $Val(\ell, i, \mathbf{x}, \mathbf{Y})$  is the argument of the activation function used to compute some  $(\mathbf{v}_{\ell})_i$ .

Note that sets  $\mathcal{X}_{\ell,i}$  are nonempty, and they can be infinite. However, Theorem 8 shows that  $\mathcal{X}_{\ell,i}$  can be enumerated as a countable, monotonically increasing sequence of numbers. This is important because it shows that the notion of a least nonzero element of  $\mathcal{X}_{\ell,i}$  is correctly defined. In the following, for each  $\alpha \in \mathbb{R}$ , let  $\mathcal{X}_{\ell,i}^{>\alpha} = \{\alpha' \in \mathcal{X}_{\ell,i} \mid \alpha' > \alpha\}.$ 

**Theorem 8.** Each set  $\mathcal{X}_{\ell,i}$  satisfies  $\mathcal{X}_{\ell,i} \subseteq \mathbb{R}_0^+$ , and, for each  $\alpha \in \mathbb{R}$ , set  $\mathcal{X}_{\ell,i} \setminus \mathcal{X}_{\ell,i}^{>\alpha}$  is finite.

Theorem 8 ensures that, for each  $\alpha \in \mathbb{R}$ , set  $\mathcal{X}_{\ell i}^{>\alpha}$  is either empty or it contains a smallest number strictly larger than  $\alpha$ . The proof uses the fact that the activation function  $\sigma$  is unbounded. We are now ready to define the capacity of  $\mathcal{N}$ .

**Definition 9.** The capacity of each layer  $\ell$  of  $\mathcal{N}$  is defined in Algorithm 1. Moreover, the capacity of N is defined as  $C_{\mathcal{N}} = \max\{C_1, \dots, C_L\}.$ 

Sets  $\mathcal{X}_{\ell,i}$  can be infinite, so Algorithm 1 can perhaps be better understood as inductively defining sequences of numbers  $\alpha_{\ell}$ ,  $\beta_{\ell}$ ,  $C_{\ell}$  and so on. However, in Section 4.3 we show that the smallest positive elements of  $\mathcal{X}_{\ell,i}$  can in fact be computed, which justifies our usage of the term 'algorithm'.

Theorem 10 shows that, in each layer of  $\ell$ , every  $k_{\ell}$  that is larger than  $C_{\ell}$  can be replaced by  $C_{\ell}$  without affecting the result of applying  $\mathcal{N}$  to any dataset.

**Theorem 10.** Let  $\mathcal{N}'$  be the  $(Col, \delta)$ -GNN obtained from  $\mathcal{N}$  by replacing  $k_{\ell}$  with  $C_{\ell}$  for each  $1 \leq \ell \leq L$ . Then,  $T_{\mathcal{N}}(D) = T_{\mathcal{N}'}(D)$  for each  $(Col, \delta)$ -dataset D.

Theorem 10 can be intuitively understood as follows. Let  $\mathbf{v}_{\lambda_{\ell}}$  and  $\mathbf{v}_{\lambda'_{\ell}}$  be vectors labelling a vertex v in layer  $\ell$  when  $T_{\mathcal{N}}$  and  $T_{\mathcal{N}'}$  are applied to some D. We prove the theorem by showing that either  $(\mathbf{v}_{\lambda_{\ell}})_i = (\mathbf{v}_{\lambda'_{\ell}})_i$  or  $(\mathbf{v}_{\lambda_{\ell}})_i > (\mathbf{v}_{\lambda'_{\ell}})_i \geq \alpha_{\ell}$  for each layer  $\ell \geq \ell_{st}$ , where  $\ell_{st}$  is either the layer where Algorithm 1 performs an early return (via line 7) or 0 if this does not happen. Indeed, assume that  $cls((\mathbf{v}_{\lambda_L})_i) = 1$  for some v. If  $\mathbf{A}_L$  and all  $\mathbf{B}_L^c$  contain only zeros, or if all  $\mathcal{X}_{L,i}$  contain only zeros, then  $L = \ell_{st}$ ; no neighbours of v are needed so we can set all  $C_{\ell}$  to 0 and the equality above holds. Otherwise, cls is a threshold function, so  $(\mathbf{v}_{\lambda_L})_i \geq \alpha_L$  holds for  $\alpha_L$  the threshold of cls, and so the argument to the activation function when computing  $(\mathbf{v}_{\lambda_L})_i$ is at least  $\beta_L$ . Moreover,  $(\mathbf{v}_{\lambda_L})_i$  is produced from  $(\mathbf{v}_{\lambda_{L-1}})_i$ and the values of  $(\mathbf{u}_{\lambda_{L-1}})_j$  where *u* ranges over the neighbours of v. If we assume that  $(\mathbf{v}_{\lambda_{L-1}})_i = 0$  and that  $\epsilon_{\ell}$  is the least nonzero value that each u can contribute to  $(\mathbf{v}_{\lambda_L})_i$ , it suffices to have at least  $\lceil \frac{\beta_{\ell} - b_{\ell}}{w_{\ell} \cdot \epsilon_{\ell}} \rceil$  nonzero neighbours to reach  $\beta_L$ . Thus, we can replace  $k_{\ell}$  with this number whenever this number is smaller than  $k_{\ell}$ ; in contrast, if  $k_{\ell}$  is smaller, we need to keep  $k_\ell$  so that  $\mathcal{N}'$  does not derive any new consequences. Finally,  $\alpha_{L-1}$  is the value of  $(\mathbf{v}_{\lambda_{L-1}})_i$  in layer L-1 to which we can apply analogous reasoning.

### 4.2 Equivalence with Datalog Programs

We next show that there exists a Datalog program  $\mathcal{P}_{\mathcal{N}}$  that is equivalent to  $\mathcal{N}$  in the sense described in Definition 4. Towards this goal, in Definition 11 we capture the syntactic structure of the rules in  $\mathcal{P}_{\mathcal{N}}$  as rules of form (25) where  $\varphi$  is a *tree-like* formula for x. To understand the intuition, assume that we construct from  $\varphi$  a graph whose vertices are the variables in  $\varphi$ , and where a directed edge from x to y is introduced for each  $E^{c}(x, y)$  in  $\varphi$ ; then, such graph must be a directed tree. Moreover, if variable x has children  $y_1$  and  $y_2$  in this graph, then  $\varphi$  is allowed to contain inequalities of the form  $y_1 \not\approx y_2$ , which provide  $\varphi$ with a limited capability for counting; for example, formula  $E^{c}(x, y_{1}) \wedge E^{c}(x, y_{2}) \wedge y_{1} \not\approx y_{2}$  is true precisely for those values of x that are connected via the  $E^c$  predicate to at least two distinct constants. We also introduce intuitive notions of a fan-out (i.e., the number of children) and depth of a variable. Tree-like formulas contain all concepts of the ALCO. description logic (Baader et al. 2007) constructed from  $\top$ , atomic concepts, and concepts of the form  $\geq nR.C$  and  $C_1 \sqcap C_2$ ; however, our definition also allows for formulas such as  $E^c(x, y_1) \wedge E^c(x, y_2) \wedge U(y_1) \wedge y_1 \not\approx y_2$ , which do not correspond to the translation of ALCQ concepts.

**Definition 11.** A tree-like formula for a variable *is defined inductively as follows.* 

- For each variable x, formula  $\top$  is tree-like for x.
- For each variable x and each unary predicate U, atom U(x) is tree-like for x.
- For each variable x and all tree-like formulas  $\varphi_1$  and  $\varphi_2$  for x that share no variables other than x, formula  $\varphi_1 \wedge \varphi_2$  is tree-like for x.

• For each variable x, each binary predicate  $E^c$ , and all tree-like formulas  $\varphi_1, \ldots, \varphi_n$  for distinct variables  $y_1, \ldots, y_n$  where no  $\varphi_i$  contains x and no  $\varphi_i$  and  $\varphi_j$  with  $i \neq j$  share a variable, formula (24) is tree-like for x.

$$\bigwedge_{i=1}^{n} \left( E^{c}(x, y_{i}) \land \varphi_{i} \right) \land \bigwedge_{1 \le i < j \le n} y_{i} \not\approx y_{j}$$
(24)

Let  $\varphi$  be a tree-like formula and let x be a variable in  $\varphi$ . The fan-out of x in  $\varphi$  is the number of distinct variables  $y_i$  for which  $E^c(x, y_i)$  is a conjunct of  $\varphi$ . The depth of x is the maximal n for which there exist variables  $x_0, \ldots, x_n$  and predicates  $E^{c_1}, \ldots, E^{c_n}$  such that  $x_n = x$ and  $E^{c_i}(x_{i-1}, x_i)$  is a conjunct of  $\varphi$  for each  $1 \le i \le n$ . The depth of  $\varphi$  is the maximum depth of a variable in  $\varphi$ .

For d and f natural numbers, a tree-like formula  $\varphi$  is (d, f)-tree-like if, for each variable x in  $\varphi$ , the depth i of x is at most d and the fan-out of x is at most f(d-i). Moreover, a Datalog rule is (d, f)-tree-like if it is of form (25), where  $\varphi$  is a (d, f)-tree-like formula for x.

$$\varphi \to U(x)$$
 (25)

Note that  $\varphi$  is allowed to be  $\top$  in a rule of form (25); for example,  $\top \to U(x)$  is a valid (0,0)-tree-like rule. As explained in Section 2, when applied to a dataset D, such a rule derives U(t) for each term t occurring in D.

Now let  $\delta_{\mathcal{N}} = \max(\delta_0, \ldots, \delta_L)$ . To construct  $\mathcal{P}_{\mathcal{N}}$ , we proceed as follows: we compute  $f = |\text{Col}| \cdot \delta_{\mathcal{N}} \cdot C_{\mathcal{N}}$ , we enumerate all (L, f)-tree-like rules (up to variable renaming), and we add to  $\mathcal{P}_{\mathcal{N}}$  each such rule that is captured by  $\mathcal{N}$ . Lemma 12 shows that this latter test can, at least in principle, be operationalised. In particular, to test whether a rule  $\varphi \rightarrow U(x)$  with *n* variables is captured by  $\mathcal{N}$ , we consider each possible dataset *D* obtained from the atoms of  $\varphi$  by substituting the variables with up to *n* distinct constants, and we check whether applying  $\mathcal{N}$  to *D* derives the analogously instantiated rule head; if this is the case for all such *D*, then the rule is captured by  $\mathcal{N}$ . Tena Cucala et al. (2022) used a similar test for MGNNs, but their approach was simpler since it did not need to support inequalities. Theorem 13 then shows that program  $\mathcal{P}_{\mathcal{N}}$  is indeed equivalent to  $\mathcal{N}$ .

**Lemma 12.** Let r be a constant-free Datalog rule with head H, let V be the set of variables in r, and let A be the set of body atoms of r. Then,  $\mathcal{N}$  captures r if and only if  $H\nu \in T_{\mathcal{N}}(A\nu)$  for each substitution  $\nu : V \to S$  such that  $H\nu \in T_r(A\nu)$ , where S is a set of |V| distinct constants.

**Theorem 13.** Let  $\mathcal{P}_{\mathcal{N}}$  be the Datalog program containing, up to variable renaming, each  $(L, |\mathsf{Col}| \cdot \delta_{\mathcal{N}} \cdot C_{\mathcal{N}})$ -tree-like rule captured by  $\mathcal{N}$ , where  $\delta_{\mathcal{N}} = \max(\delta_0, \ldots, \delta_L)$ . Then,  $\mathcal{N}$  and  $\mathcal{P}_{\mathcal{N}}$  are equivalent.

To understand this result intuitively, assume that  $\mathcal{N}$  is applied to a dataset D. The fact that all rules of  $\mathcal{P}_{\mathcal{N}}$  are captured by  $\mathcal{N}$  clearly implies  $T_{\mathcal{P}_{\mathcal{N}}}(D) \subseteq T_{\mathcal{N}}(D)$ . Furthermore, by equation (3), the value of  $(\mathbf{v}_L)_i$  for some i is computed from the values of  $(\mathbf{v}_{L-1})_i$  and  $(\mathbf{u}_{L-1})_j$  for  $k \leq C_L$  distinct neighbours u of v per colour and position; but then, if t and s are terms represented by v and u, respectively, the canonical encoding ensures  $E^c(t, s) \in D$  for some  $c \in \text{Col}$ .

Also,  $(\mathbf{u}_{L-1})_i$  are computed using the neighbours of u and so on. Hence, each term w in D that can possibly influence  $\mathbf{v}_L$  must be connected in D to t by at most L such facts, so all relevant neighbours of t can be selected by a (d, f)-treelike formula. The inequalities can be used to check for the existence of at least k distinct neighbours of t in D. Now let D' be the subset of D containing precisely the facts that contribute to the value of  $(\mathbf{v}_L)_i$ . We can unfold D' into another tree-like dataset D'' that corresponds to the body of an instantiated tree-like rule r. Since the elements of all  $A_{\ell}$  and  $\mathbf{B}_{\ell}^{c}$  are nonnegative, applying  $\mathcal{N}$  to D and D'' derives the same value for  $cls((\mathbf{v}_L)_i)$ . If this value is 1, then applying the rule r to D produces the same fact as  $\mathcal{N}$ . Furthermore, by definition,  $\mathcal{N}$  captures r and so  $r \in \mathcal{P}_{\mathcal{N}}$ . Thus,  $T_{\mathcal{P}_{\mathcal{N}}}(D)$ contains all facts derived by  $\mathcal{N}$  on D.

#### Enumerating Sets $\mathcal{X}_{\ell,i}$ 4.3

The results we presented thus far show that program  $\mathcal{P}_{\mathcal{N}}$ exists, but it is not yet clear that  $\mathcal{P}_{\mathcal{N}}$  is computable: the definition of  $C_{\ell}$  in Algorithm 1 uses sets  $\mathcal{X}_{\ell,i}$ , which can be infinite. We next show that each  $\mathcal{X}_{\ell,i}$  can be enumerated algorithmically using function Next $(\ell, i, \alpha)$  from Algorithm 2 as follows: for  $\alpha$  a special symbol  $\triangleright$ , function Next $(\ell, i, \triangleright)$ returns the smallest element of  $\mathcal{X}_{\ell,i}$ ; moreover, for  $\alpha \in \mathbb{R}$ , function Next $(\ell, i, \alpha)$  returns the smallest element of  $\mathcal{X}_{\ell,i}^{>\alpha}$ if  $\mathcal{X}_{\ell,i}^{>\alpha} \neq \emptyset$ , or  $\lhd$  otherwise. For example,  $\mathsf{Next}(\ell, i, 0)$  returns the smallest nonzero element of  $\mathcal{X}_{\ell,i}$ , if one exists.

In the presentation of Algorithm 2, we use the following notation: for  $\mathbf{x}$  a vector, j an index, and v a real number,  $\mathbf{x}[i \leftarrow v]$  is the vector obtained from x by replacing its j-th component with v. The algorithm is based on the observation that, since  $A_{\ell}$  and  $B_{\ell}^{c}$  contain only nonnegative elements, and the activation function is monotonically increasing, we can enumerate the values computed by equation (3) in some  $\mathbf{v}_{\ell}$  in a monotonically increasing fashion. To achieve this, the algorithm maintains a *frontier* Fof triples  $\langle \mathbf{x}, \mathbf{Y}, z \rangle$ , each describing one way to compute a value of  $(\mathbf{v}_{\ell})_i$ : vector **x** reflects the values of  $(\mathbf{v}_{\ell-1})_i$ , the  $(Col, \ell - 1)$ -multiset family Y describes multisets Y<sup>c</sup> reflecting the values of  $(\mathbf{u}_{\ell-1})_i$ , and z is  $\mathsf{Val}(\ell, i, \mathbf{x}, \mathbf{Y})$ —that is, the argument to the activation function when computing  $(\mathbf{v}_{\ell})_i$ . The starting point for the exploration (line 8) is provided by  $\text{Start}(\ell)$ , which returns  $\mathbf{v}_{\ell}$  for a vertex v with no neighbours. To enumerate all candidate values for  $(\mathbf{v}_{\ell})_i$  in an increasing order, the algorithm selects a triple in the frontier with the smallest z (line 10), and considers ways to modify  $\mathbf{x}$  or  $\mathbf{Y}$  that increase z; each such combination is added to the frontier (lines 14, 19, and 27). Modifications involve replacing some component of x with the next component (lines 12–14), choosing some  $\mathbf{y} \in \mathbf{Y}^c$  for some  $c \in \mathsf{Col}$  and replacing some component of y with the next component (lines 16–19), or expanding some  $\mathbf{Y}^c$  with an additional vector (lines 20–27). In the latter case, if  $Start(\ell)$  contains just zeros, then adding  $Start(\ell)$  to  $\mathbf{Y}^c$  is not going to change the computed value of z so the algorithm considers vectors obtained by expanding  $\text{Start}(\ell)$  in order to allow z to increase. This process produces values of z in an increasing order and it guarantees that  $\sigma(z) \in \mathcal{X}_{\ell,i}$ . If  $\alpha = \triangleright$ , the algorithm stops Algorithm 2 Next( $\ell, i, \alpha$ )

1: if  $\ell = 0$  then if  $\alpha = \triangleright$  or  $\alpha < 0$  then return 0 2: else if  $\alpha < 1$  then return 13: 4: else return ⊲ 5: let  $\mathbf{Y}_{\emptyset}$  be such that  $\mathbf{Y}_{\emptyset}^{c} = \emptyset$  for each  $c \in \mathsf{Col}$ 6:  $z := Val(\ell, i, Start(\ell), \mathbf{Y}_{\emptyset})$ 7: if  $\alpha = \triangleright$  then return  $\sigma(z)$ 8:  $F := \{ \langle \mathsf{Start}(\ell), \mathbf{Y}_{\emptyset}, z \rangle \}$ 9: while  $F \neq \emptyset$  do choose and remove  $\langle \mathbf{x}, \mathbf{Y}, z \rangle$  in F with least z 10: if  $\sigma(z) > \alpha$  then return  $\sigma(z)$ 11: 12: for  $\mathbf{x}' \in \mathsf{Expand}(\ell, \mathbf{x})$  do 13:  $z' := \mathsf{Val}(\ell, i, \mathbf{x}', \mathbf{Y})$ 14: if z' > z then add  $\langle \mathbf{x}', \mathbf{Y}, z' \rangle$  to F for  $c \in \operatorname{Col} \operatorname{do}$ 15: for  $\mathbf{y} \in \mathbf{Y}^c$  and  $\mathbf{y}' \in \mathsf{Expand}(\ell, \mathbf{y})$  do 16:  $\mathbf{\tilde{Y}'} := \mathbf{Y} \text{ and } \mathbf{\tilde{Y}'^c} := (\mathbf{Y'^c} \setminus \{\mathbf{\tilde{y}}\}) \cup \{\mathbf{y'}\}$ 17:  $z' := \mathsf{Val}(\ell, i, \mathbf{x}, \mathbf{Y}')$ 18: if z' > z then add  $\langle \mathbf{x}, \mathbf{Y}', z' \rangle$  to F19: if  $Start(\ell)$  contains a nonzero then 20:  $V := \{\mathsf{Start}(\ell)\}$ 21: 22: else  $V := \mathsf{Expand}(\ell, \mathsf{Start}(\ell))$ 23: for  $\mathbf{y}' \in V$  do 24:  $\mathbf{\hat{Y}}' := \mathbf{Y}$  and  $\mathbf{Y}'^c := \mathbf{Y}'^c \cup \{\mathbf{y}'\}$ 25:  $z' := \mathsf{Val}(\ell, i, \mathbf{x}, \mathbf{Y}')$ 26: if z' > z then add  $\langle \mathbf{x}, \mathbf{Y}', z' \rangle$  to F 27: 28: **return** ⊲ 29: function  $Start(\ell)$ **return** the vector **x** of dimension  $\delta_{\ell-1}$  where 30:  $(\mathbf{x})_j = \mathsf{Next}(\ell - 1, j, \rhd) \text{ for } 1 \le j \le \delta_{\ell - 1}$ 

31: function Expand( $\ell$ , **v**)

- 32:  $V := \emptyset$
- for  $1 \leq j \leq \delta_{\ell-1}$  do 33:

34: 
$$v' := \operatorname{Next}(\ell - 1, j, (\mathbf{v})_j)$$
  
35: if  $v' \neq \lhd$  then  $V := V \cup \{\mathbf{v}[j \leftarrow v']\}$ 

return V 36:

when the first such value is produced (line 7). For  $\alpha \in \mathbb{R}$ , Theorem 8 guarantees that set  $\mathcal{X}_{\ell,i} \setminus \mathcal{X}_{\ell,i}^{>\alpha}$  is finite; since F is extended only if the value of z increases, either F eventually becomes empty or  $\sigma(z)$  exceeds  $\alpha$  so the algorithm terminates (line 11 or 28). Theorem 14 captures the formal properties of the algorithm.

Theorem 14. Algorithm 2 terminates on all inputs. Moreover, for  $0 \leq \ell \leq L$  and  $1 \leq i \leq \delta_{\ell}$ ,

- Next $(\ell, i, \triangleright)$  returns the smallest element of  $\mathcal{X}_{\ell,i}$ , and
- for each  $\alpha \in \mathbb{R}$ ,  $Next(\ell, i, \alpha)$  returns  $\triangleleft$  if  $\mathcal{X}_{\ell,i}^{>\alpha} = \emptyset$ , and otherwise it returns the smallest element of  $\chi^{>\alpha}_{\ell_i}$ .

The complexity of Algorithm 14 depends on the number of recursive calls to Next, which in turn depends on the matrices of  $\mathcal{N}$ . We leave investigating this issue to future work.

(

# 5 Limiting Aggregation to Max

In this section we study the expressivity of *monotonic max GNNs*, which follow the same restrictions as monotonic max-sum GNNs but additionally allow only for the max aggregation function. Theorem 16 shows that each such GNN corresponds to a Datalog program without inequalities. Consequently, monotonic max GNNs cannot count the connections of a constant in a dataset.

**Definition 15.** A monotonic max  $(Col, \delta)$ -GNN *is a monotonic max-sum GNN that uses the* max-1-sum *aggregation function in all layers.* 

**Theorem 16.** For each monotonic max  $(Col, \delta)$ -GNN  $\mathcal{N}$  with L layers, let  $\delta_{\mathcal{N}} = \max(\delta_0, \ldots, \delta_L)$ , and let  $\mathcal{P}_{\mathcal{N}}$  be the Datalog program containing up to variable renaming each  $(L, |Col| \cdot \delta_{\mathcal{N}})$ -tree-like rule without inequalities captured by  $\mathcal{N}$ . Then,  $\mathcal{N}$  and  $\mathcal{P}_{\mathcal{N}}$  are equivalent.

Tena Cucala et al. (2022) presented a closely related characterisation for MGNNs, and the main difference is that we use the canonical encoding. The latter allows us to describe the target Datalog class more precisely, which in turn allows us to prove the converse: each Datalog program with only tree-like rules and without inequalities is equivalent to a monotonic max GNN.

In what follows, we fix a program  $\mathcal{P}$  consisting of (d, f)tree-like rules without inequalities. Recall that the signature of  $\mathcal{P}$  consists of unary predicates  $U_1, \ldots, U_{\delta}$  and binary predicates  $E^c$  for  $c \in \text{Col.}$  Now let  $\tau_1, \ldots, \tau_n$  be a sequence containing up to variable renaming each (d, f)-tree-like formula for variable x without inequalities ordered by increasing depth—that is, for all i < j, the depth of  $\tau_i$  is less than or equal to the depth of  $\tau_j$ . Each  $\tau_i$  can be written as

$$\tau_i = \varphi_{i,0} \wedge \bigwedge_{k=1}^{m_i} \left( E^{c_k}(x, y_k) \wedge \varphi_{i,k} \right), \tag{26}$$

where  $\varphi_{i,0}$  is a conjunction of unary atoms using only variable x, each  $\varphi_{i,k}$  with  $1 \le k \le m_i$  is a (d-1, f)-tree-like formula for  $y_k$ , and, for all  $1 \le k < k' \le m_i$ , formulas  $\varphi_{i,k}$  and  $\varphi_{i,k'}$  do not have variables in common. Note that formulas  $\varphi_{i,k}$  can be  $\top$ , and that colours  $c_k$  need not be distinct.

We define  $\mathcal{N}_{\mathcal{P}}$  as the monotonic max (Col,  $\delta$ )-GNN of form (2) satisfying the following conditions. The number of layers is L = d + 2, the activation function is ReLU, and the classification function cls is the step function with threshold 1. For  $1 \leq \ell < L$ , dimension  $\delta_{\ell}$  is defined as the number of formulas in the above sequence of depth at most  $\ell - 1$ . The elements of  $\mathbf{A}_{\ell}$ ,  $\mathbf{B}_{\ell}^{c}$ , and  $\mathbf{b}_{\ell}$  are defined as follows, for  $c \in \text{Col}$ ,  $1 \leq \ell \leq L$ ,  $1 \leq i \leq \delta_{\ell}$ , and  $1 \leq j \leq \delta_{\ell-1}$ .

$$(\mathbf{A}_{\ell})_{i,j} = \begin{cases} 1 & \text{if} \\ \bullet \ \ell = 1 \text{ and } \tau_i \text{ contains } U_j(x); \text{ or} \\ \bullet \ 2 \le \ell < L \text{ and} \\ -1 \le i \le \delta_{\ell-1} \text{ and } i = j, \text{ or} \\ -\delta_{\ell-1} < i \le \delta_{\ell} \text{ and } \varphi_{i,0} = \tau_j; \text{ or} \\ \bullet \ \ell = L \text{ and } \mathcal{P} \text{ contains rule} \\ \tau_j \to U_i(x) \text{ up to variable renaming;} \\ 0 \text{ otherwise.} \end{cases}$$

$$\mathbf{B}_{\ell}^{c})_{i,j} = \begin{cases} 1 & \text{if } 2 \leq \ell < L \text{ and there exists } 1 \leq k \leq m_{i} \\ & \text{such that } c = c_{k} \text{ and } \varphi_{i,k} \text{ and } \tau_{j} \\ & \text{are equal up to variable renaming;} \\ 0 & \text{otherwise.} \end{cases}$$

$$(\mathbf{b}_{\ell})_i = \begin{cases} 1 - \sum_{j=1}^{\delta_{\ell}-1} ((\mathbf{A}_{\ell})_{i,j} + \sum_{c \in \mathsf{Col}} (\mathbf{B}_{\ell}^c)_{i,j}) & 1 \leq \ell < L \text{ and} \\ \delta_{\ell-1} < i \leq \delta_{\ell}; \\ 0 & \text{otherwise.} \end{cases}$$

To understand the intuition behind the construction of  $\mathcal{N}_{\mathcal{P}}$ , assume that  $\mathcal{N}_{\mathcal{P}}$  is applied to a dataset D, and consider a vector  $\mathbf{v}_{\ell}$  labelling in layer  $\ell$  a vertex corresponding to some term t of D. Then, the *i*-th component of  $\mathbf{v}_{\ell}$  is paired with formula  $\tau_i$  from the above enumeration, and it indicates whether it is possible to evaluate  $\tau_i$  over D by mapping variable x to t. This is formally captured by Lemma 17. To ensure that  $\mathcal{N}_{\mathcal{P}}$  and  $\mathcal{P}$  are equivalent, layer L of  $\mathcal{N}_{\mathcal{P}}$  simply realises a disjunction over all rules in the program.

**Lemma 17.** For each  $(Col, \delta)$ -dataset D, layer  $1 \le \ell < L$ of  $\mathcal{N}_{\mathcal{P}}$ , position  $1 \le i \le \delta_{\ell}$ , and term t in D, and for  $\mathbf{v}_{\ell}$ the labelling of the vertex corresponding to t when  $\mathcal{N}_{\mathcal{P}}$  is applied to the canonical encoding of D,

- $(\mathbf{v}_{\ell})_i = 1$  if there exists a substitution  $\nu$  mapping x to t such that  $D \models \tau_i \nu$ , and
- $(\mathbf{v}_{\ell})_i = 0$  otherwise.

Note that each  $\delta_{\ell}$  with  $1 \leq \ell < L$  is determined by the number of (d, f)-tree-like formulas of depth  $\ell - 1$ , and that  $\delta_{L-1}$  is the largest such number. We next determine an upper bound on  $\delta_{L-1}$ . By Definition 11, the fan-out of a variable of depth i is at most f(d-i). The number of variables of depth i is at most the number of variables of depth i-1 times the fan-out of each variable, which is  $f^i \cdot d \dots (d-i+1)$  and is bounded by  $f^i \cdot d!$ . By adding up the contribution for each depth, there are at most  $f^d \cdot (d+1)!$  variables. Each variable is labelled by one of the  $2^{\delta}$  conjunctions of depth zero, and each non-root variable is connected by one of the |Col| predicates to its parent. Hence, there are at most  $(|\text{Col}| \cdot 2^{\delta})^{f^d \cdot (d+1)!}$  tree-like formulas.

**Theorem 18.** Program  $\mathcal{P}$  and GNN  $\mathcal{N}_{\mathcal{P}}$  are equivalent, and moreover  $\delta_{L-1} \leq (|\mathsf{Col}| \cdot 2^{\delta})^{f^d \cdot (d+1)!}$ .

## 6 Conclusion

We have shown that each monotonic max-sum GNN (i.e., a GNN that uses max and sum aggregation functions and satisfies certain properties) is equivalent to a Datalog program with inequalities in the sense that applying the GNN or a single round of the rules of the program to any dataset produces the same result. We have also sharpened this result to monotonic max GNNs and shown the converse: each tree-like Datalog program without inequalities is equivalent to a monotonic max GNN. We see many avenues for future work. First, we aim to completely characterise monotonic max-sum GNNs. Second, we intend to implement rule extraction. Third, we shall investigate the empirical performance of monotonic max-sum GNNs on tasks other than link prediction, such as node classification.

# Acknowledgements

This work was supported by the SIRIUS Centre for Scalable Data Access (Research Council of Norway, project number 237889), and the EPSRC projects ConCur (EP/V050869/1), UK FIRES (EP/S019111/1), and AnaLOG (EP/P025943/1). For the purpose of Open Access, the author has applied a CC BY public copyright licence to any Author Accepted Manuscript (AAM) version arising from this submission.

## References

Abiteboul, S.; Hull, R.; and Vianu, V. 1995. *Foundations of Databases*. Addison Wesley.

Baader, F.; Calvanese, D.; McGuinness, D.; Nardi, D.; and Patel-Schneider, P. F., eds. 2007. *The Description Logic Handbook: Theory, Implementation and Applications*. Cambridge University Press, 2nd edition.

Bader, S.; Hitzler, P.; Hölldobler, S.; and Witzel, A. 2007. A Fully Connectionist Model Generator for Covered First-Order Logic Programs. In *Proc. IJCAI*, 666–671.

Bader, S.; d'Avila Garcez, A. S.; and Hitzler, P. 2005. Computing First-Order Logic Programs by Fibring Artificial Neural Networks. In *Proc. FLAIRS*, 314–319. AAAI Press.

Barceló, P.; Kostylev, E. V.; Monet, M.; Pérez, J.; Reutter, J. L.; and Silva, J. P. 2020. The Logical Expressiveness of Graph Neural Networks. In *Proc. ICLR*.

Campero, A.; Pareja, A.; Klinger, T.; Tenenbaum, J.; and Riedel, S. 2018. Logical rule induction and theory learning using neural theorem proving. *CoRR* abs/1809.02193.

Dantsin, E.; Eiter, T.; Gottlob, G.; and Voronkov, A. 2001. Complexity and expressive power of logic programming. *ACM Comput. Surv.* 33(3):374–425.

Dong, H.; Mao, J.; Lin, T.; Wang, C.; Li, L.; and Zhou, D. 2019. Neural Logic Machines (Poster). In *Proc. ICLR*.

Hölldobler, S.; Kalinke, Y.; and Störr, H.-P. 1999. Approximating the Semantics of Logic Programs by Recurrent Neural Networks. *Applied Intelligence* 11(1):45–58.

Huang, X.; Orth, M. A. R.; Ceylan, İ. İ.; and Barceló, P. 2023. A theory of link prediction via relational weisfeiler-leman. *CoRR* abs/2302.02209.

Ioannidis, V. N.; Marques, A. G.; and Giannakis, G. B. 2019. A recurrent graph neural network for multi-relational data. In *Proc. ICASSP*, 8157–8161. IEEE.

Kipf, T. N., and Welling, M. 2017. Semi-supervised classification with graph convolutional networks. In *Proc. ICLR*.

Liu, S.; Cuenca Grau, B.; Horrocks, I.; and Kostylev, E. V. 2021. INDIGO: gnn-based inductive knowledge graph completion using pair-wise encoding. In *Proc. NeurIPS*, 2034–2045.

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

Motik, B.; Cuenca Grau, B.; Horrocks, I.; Wu, Z.; Fokoue, A.; and Lutz, C. 2012. Owl 2 web ontology language: Profiles (2nd edition). W3C Recommendation.

Pérez, J.; Arenas, M.; and Gutierrez, C. 2009. Semantics and complexity of SPARQL. *ACM Trans. Database Syst.* 34(3):16:1–16:45.

Pflueger, M.; Tena Cucala, D. J.; and Kostylev, E. V. 2022. GNNQ: A neuro-symbolic approach to query answering over incomplete knowledge graphs. In *Proc. ISWC*, volume 13489 of *Lecture Notes in Computer Science*, 481–497. Springer.

Qu, M.; Bengio, Y.; and Tang, J. 2019. GMNN: graph markov neural networks. In *Proc. ICML*, volume 97 of *Proceedings of Machine Learning Research*, 5241–5250.

Rocktäschel, T., and Riedel, S. 2017. End-to-end Differentiable Proving. In *Proc. NeurIPS*, 3788–3800.

Sadeghian, A.; Armandpour, M.; Ding, P.; and Wang, D. Z. 2019. DRUM: End-To-End Differentiable Rule Mining On Knowledge Graphs. In *Proc. NeurIPS*, 15321–15331.

Schlichtkrull, M. S.; Kipf, T. N.; Bloem, P.; van den Berg, R.; Titov, I.; and Welling, M. 2018. Modeling relational data with graph convolutional networks. In *Proc. ESWC*, volume 10843 of *LNCS*, 593–607. Springer.

Sourek, G.; Zelezný, F.; and Kuzelka, O. 2021. Beyond graph neural networks with lifted relational neural networks. *Mach. Learn.* 110(7):1695–1738.

Tena Cucala, D.; Cuenca Grau, B.; Kostylev, E. V.; and Motik, B. 2022. Explainable GNN-Based Models over Knowledge Graphs. In *Proc. ICLR*.

Tena Cucala, D.; Grau, B. C.; Motik, B.; and Kostylev, E. V. 2023. On the Correspondence Between Monotonic Max-Sum GNNs and Datalog. *CoRR* abs/2305.18015.

Tena Cucala, D.; Cuenca Grau, B.; and Motik, B. 2022. Faithful Approaches to Rule Learning. In *Proc. KR*, 484–493.

Teru, K. K.; Denis, E. G.; and Hamilton, W. L. 2020. Inductive relation prediction by subgraph reasoning. In *Proc. ICML*, volume 119 of *Proceedings of Machine Learning Research*, 9448–9457. PMLR.

Yang, Z.; Cohen, W. W.; and Salakhutdinov, R. 2016. Revisiting semi-supervised learning with graph embeddings. In *Proc. ICML*, volume 48 of *JMLR Workshop and Conference Proceedings*, 40–48.

Yang, F.; Yang, Z.; and Cohen, W. W. 2017. Differentiable Learning of Logical Rules for Knowledge Base Reasoning. In *Proc. NeurIPS*, 2319–2328.

Zhang, M., and Chen, Y. 2018. Link prediction based on graph neural networks. In *Proc. NeurIPS*, 5171–5181.