Machine Learning (2022) 111:575–623
https://doi.org/10.1007/s10994-021-06090-8

Inclusion of domain-knowledge into GNNs using mode-directed inverse entailment

Tirtharaj Dash[1,2] · **Ashwin Srinivasan**[1,2] · **A. Baskar**[2]

Received: 25 May 2021 / Revised: 17 August 2021 / Accepted: 27 September 2021 /
Published online: 18 November 2021

Abstract
We present a general technique for constructing Graph Neural Networks (GNNs) capable of using multi-relational domain knowledge. The technique is based on mode-directed inverse entailment (MDIE) developed in Inductive Logic Programming (ILP). Given a data instance e and background knowledge B, MDIE identifies a most-specific logical formula $\bot_B(e)$ that contains all the relational information in B that is related to e. We represent $\bot_B(e)$ by a "bottom-graph" that can be converted into a form suitable for GNN implementations. This transformation allows a principled way of incorporating generic background knowledge into GNNs: we use the term 'BotGNN' for this form of graph neural networks. For several GNN variants, using real-world datasets with substantial background knowledge, we show that BotGNNs perform significantly better than both GNNs without background knowledge and a recently proposed simplified technique for including domain knowledge into GNNs. We also provide experimental evidence comparing BotGNNs favourably to multi-layer perceptrons that use features representing a "propositionalised" form of the background knowledge; and BotGNNs to a standard ILP based on the use of most-specific clauses. Taken together, these results point to BotGNNs as capable of combining the computational efficacy of GNNs with the representational versatility of ILP.

Keywords Neuro-symbolic learning · Inductive logic programming · Mode-directed inverse entailment · Graph neural networks · Background knowledge

Editors: Nikos Katzouris, Alexander Artikis, Luc De Raedt, Artur d'Avila Garcez, Ute Schmid, Sebastijan Dumančić, Jay Pujara.

✉ Tirtharaj Dash
tirtharaj@goa.bits-pilani.ac.in

Ashwin Srinivasan
ashwin@goa.bits-pilani.ac.in

A. Baskar
abaskar@goa.bits-pilani.ac.in

[1] APPCAIR, BITS Pilani, K.K. Birla Goa Campus, Goa 403726, India

[2] Department of CS and IS, BITS Pilani, K.K. Birla Goa Campus, Goa 403726, India

1 Introduction

Scientific progress is largely a cumulative enterprise: hypotheses are devised and experiments conducted based on what is already known in the field. Recent ambitious developments (see the Nobel-Turing Grand Challenge Kitano, 2016) seek to accelerate significantly the process of scientific discovery, by automating the conjectures-and-refutations part of the scientific approach. At the heart of the approach is the use of Machine Learning (ML) methods to generate hypotheses about the data (we are now in the 3rd generation of such "robot scientists" (King et al., 2009), which are being used to hypothesise candidate drugs for tropical diseases like malaria: (Williams et al., 2015). The use of domain knowledge is a necessary part of such ML methods. Indeed, a recent extensive report on AI for Science (Stevens et al., 2020) has listed the inclusion of domain knowledge as the first of 3 Grand Challenges for ML and AI:

> "ML and AI are generally domain-agnostic ...Off-the-shelf practice treats [each of these] datasets in the same way and ignores domain knowledge that extends far beyond the raw data itself—such as physical laws, available forward simulations, and established invariances and symmetries—that is readily available ...Improving our ability to systematically incorporate diverse forms of domain knowledge can impact every aspect of AI, from selection of decision variables and architecture design to training data requirements, uncertainty quantification, and design optimization."

ML methods such as Deep Neural Networks (DNNs) have been shown to be extremely successful at tackling prediction problems across a range of domains such as image classification (Krizhevsky et al., 2017), machine translation (Wu et al., 2016), audio generation (Oord et al., 2016), visual reasoning (Johnson et al., 2017), etc. This has largely been possible due to: (a) the availability and easy access to large amount of data, which can be represented as numeric tensors; (b) the availability of computational hardware to allow the massively parallel computations required for large-scale neural learning. The capacity to learn from large amounts of vectorised data—a welcome development in itself—does pose some issues for a class of real-world problems, which includes many concerned with scientific discovery. These issues are: (a) The available data is structured: often represented as graphs of entities and their relationships; (b) The available data is scarce, with instances ranging from a few 10 s to a few 100 s to 1000 s. What is available in compensation, however, is a large amount of domain-knowledge, that can act as prior information. Examples of such problems abound in the natural sciences, medicine, and in the social sciences involving human studies.

There have been several proposals for a kind of DNNs devised specifically to deal with graph-structured data, called Graph Neural Networks (GNNs: Wu et al., 2020). The usual approach of *transfer learning* could alleviate the problem of learning from small amount of data if the domains of the source- and the target-problem are closely related. The other well-established route for dealing with small amounts of data involves the use of prior domain knowledge. Examples of problems with small amounts of observational data, but with large amount of compensatory prior-knowledge abound in the natural sciences, medicine, and in the social sciences involving human studies. However, general-purpose ways of incorporating such knowledge into neural networks remains elusive. In contrast, Symbolic machine learning techniques such as Inductive Logic Programming (ILP: Muggleton & De Raedt, 1994) have developed generic techniques for incorporating background knowledge—albeit encoded as logical statements—into the model-construction process.

This has an immediate importance to problems of scientific discovery, given the historical focus of scientific disciplines on mathematical and logical models: as a consequence, a substantial amount of what is known can be codified in some logical form. Further, the logical representation used by ILP systems is sufficiently expressive for representing scientific domain knowledge. However, the subsequent model-construction process can be computationally expensive, requiring a combinatorial search that cannot exploit the recent developments in specialised hardware and software libraries. In this paper, we adopt the key technique used to incorporate domain knowledge by one of the most successful form of ILP, namely, that based on mode-directed inverse entailment (MDIE: Muggleton, 1995). This form of ILP usually involves a *saturation* procedure, which efficiently identifies all the relations entailed by the domain knowledge for a specific data instance. This maximal-set of relations—called a *bottom clause*—is then used by an ILP engine to find useful logical explanations for the data. Here, we develop a corresponding saturation procedure for GNNs, which results in a maximally-specific *bottom graph*, which is then used for subsequent GNN model construction. The main contributions of this paper are as follows:

- To the field of graph neural networks, the paper proposes a systematic technique for incorporating symbolic domain-knowledge into GNNs.
- To the field of neuro-symbolic learning, the paper provides substantial empirical evidence using over 70 real-world datasets and domain-knowledge consisting over hundreds of symbolic relations that the incorporation of symbolic domain knowledge into graph-based neural networks can make a significant difference to their predictive performance.

The rest of the paper is organised as follows. We provide a brief specification and a general working principle of GNNs in Sect. 2. The basic details of saturation as used in MDIE are in Sect. 3. Our adaptation of the saturation step to construct bottom graphs is in Sect. 4. Section 5 contains an empirical evaluation of BotGNNs. We outline some of the related works in Sect. 6 providing some relevant methods for incorporating domain-knowledge into deep neural networks. Section 7 concludes the paper. The Appendices contain some conceptual, implementation- and application-related details relevant to the content in the main body of the paper.

2 Graph neural networks (GNNs)

GNNs are a class of deep neural networks suitable for learning from graph-structured data. GNNs were first introduced in Gori et al., (2005) and are recently being popularised by their use in molecular property prediction problems (Gilmer et al., 2017). In this section, we provide a general working principle of GNNs concerned with graph classification. We assume that the reader is familiar with the basics of graphs, primarily directed and undirected graphs, labelled graphs, the role of a neighbourhood function in a graph, etc. However, these details are very well discussed in Dash et al., (2021c, Sect. 2).

GNNs are concerned with labelled graphs, that is, each vertex (and each edge) of a graph are associated with a numeric feature-vector, essentially describing some properties of that vertex (or that edge). Here it suffices to state that the defining property of a GNN is that it uses some form of neural message-passing in which messages (in vector form) are exchanged between vertices of a graph and updated using a neural network (Gilmer

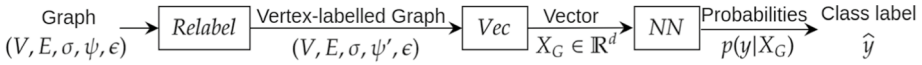


Fig. 1 A diagrammatic representation of graph classification using a GNN. Graphs are of tuples of the form $(V, E, \sigma, \psi, \epsilon)$, where V is a set of vertices; E is a set of edges; σ is some neighbourhood function; ψ is a vertex-labelling; and ϵ is an edge-labelling. Often σ is left out, and derived from the edges in E. Also, for GNNs ψ is a mapping from vertices to feature-vectors. Many GNN implementations, including the ones used in experiments here, assume the graph to be undirected and ignore the edge-labelling in ϵ

et al., 2017). For this purpose, a GNN treats the features associated with a labelled graph as 'messages' and, in the message-passing process, it updates these messages. The message-passing process is conceptually implemented by using a function called *Relabel* (this is defined in Dash et al., 2021c, Definition 4). This involves an iterative update of the vertex- (and edge-) labels. The output of the function is a relabelled graph, where the vertex- (and edge-) labels are updated.

In our present work, we are concerned with problems involving classification of (molecular) graphs. This requires a graph-level representation (also called graph-embedding Hamilton, 2020), meaning, every input graph is encoded as a d-dimensional feature vector (in $\mathbb{R}^d$). This is conceptually implemented using a function called *Vec* (refer Dash et al., 2021c, Definition 5) that vectorises a relabelled graph. This feature vector is then input to a (multi-layered) neural network, denoted by a function *NN* that maps the feature vector to a set of class-labels. The whole pipeline of graph classification is shown in Fig. 1. For completeness of presentation, we now describe how these three functions are related to the general working principle of GNNs.

2.1 General working principle of GNNs

Let $G = (V, E, \sigma, \psi, \epsilon)$ denote a graph where V is a set of vertices; E is a set of edges; σ is some neighbourhood function; ψ is a vertex-labelling; and ϵ is an edge-labelling. As mentioned earlier, we are concerned with graph classification problems. That is, given a graph G, a GNN predicts its class-label.

In a graph G, let X_v denote a vector that represents the initial labelling (ψ) of a vertex $v \in V$. That is, X_v is the feature-vector associated with the vertex v. The relabelling function $Relabel : (V, E, \sigma, \psi, \epsilon) \mapsto (V, E, \sigma, \psi', \epsilon)$ (iteratively) updates the labelling of the vertices in G. This process involves two procedures: (a) AGGREGATE: for every vertex, this procedure aggregates the information from neighboring vertices; and (b) COMBINE: this procedure updates the label of the vertex by combining its present label with its neighbors'. Mathematically, at some iteration k, the labelling of a vertex v (denoted by h_v) is updated as:

$$a_v^{(k)} = \mathsf{AGGREGATE}^{(k)}\left(\left\{h_u^{(k-1)} : u \in \mathcal{N}(v)\right\}\right),$$
$$h_v^{(k)} = \mathsf{COMBINE}^{(k)}\left(h_v^{(k-1)}, a_v^{(k)}\right)$$

where, $\mathcal{N}(v)$ denotes the set of vertices adjacent to v. Initially (at $k = 0$), $h_v^{(0)} = X_v$.

The vectorisation function $Vec : (V, E, \sigma, \psi', \epsilon) \mapsto X_G$, where $X_G \, \epsilon \, \mathbb{R}^d$ constructs a vector representation of the entire graph (also called the graph embedding). This step is carried out after the representations of all the vertices are relabelled by some iterations over AGGREGATE and COMBINE. The vectorised representation of the entire graph can be

obtained using a READOUT procedure that aggregates vertex features from the final iteration ($k = K$):

$$h_G = \text{READOUT}\left(\left\{h_v^{(K)} \mid v \in G\right\}\right)$$

In practice, AGGREGATE and COMBINE procedures are implemented using graph convolution and pooling operations. The READOUT procedure is usually implemented using a global or hierarchical pooling operation (Xu et al., 2019). Variants of GNNs result from modifications to these 3 procedures: AGGREGATE, COMBINE and READOUT.

2.2 Note on GNN variants used in this paper

In our present work, the GNN variants considered are a result of different graph convolution methods (we refer the reader to Appendix B for the mathematical aspects of these different approaches): (1) spectral graph convolution (Kipf & Welling, 2017), (2) multistage graph convolution (Morris et al., 2019), (3) graph convolution with attention (Veličković et al., 2018), (4) simple-and-aggregate graph convolution (Hamilton et al., 2017), and (5) graph convolution with auto-regressive moving average (Bianchi et al., 2021). In addition to the graph convolution methods mentioned above, we adopt the method of graph pooling with structural-attention (Lee et al., 2019) to apply down-sampling to graphs. We use the hierarchical graph-pooling approach proposed by Cangea et al.,, (2018) to implement the READOUT procedure that outputs a fixed-length representation for an input graph. This representation is then input to a multilayer perceptron (MLP) that outputs a class-label.

At this point, the complete architectural specifics of the GNN including details on various hyperparameters are not relevant. We defer these details to Sect. 5.3. We refer the reader to Appendix B for a detailed mathematical description on the five variants of graph neural networks, where we describe how the graph-convolution and graph-pooling methods are implemented followed by an elaborate description on the construction of graph-representation using the hierarchical pooling approach.

3 Mode-directed inverse entailment

Mode-directed Inverse Entailment (MDIE) was introduced by Stephen Muggleton in Muggleton (1995), as a technique for constraining the search for explanations for data in Inductive Logic Programming (ILP). For this paper, it is sufficient to focus on variants of ILP that conforms to the following input-output requirements:[1]

Given: (i) B, a set of clauses (constituting background- or domain-) knowledge; (ii) a set of clauses $E^+ = \{p_1, p_2, \ldots, p_N\}$ ($N > 0$), denoting a conjunction of "positive examples"; and (iii) a set of clauses $E^- = \{\overline{n_1}, \overline{n_2}, \ldots, \overline{n_M}\}$ ($M \geq 0$), denoting a conjunction of "negative examples", s.t.

Prior Necessity: $B \nvDash E^+$

[1] In the following, clauses will usually be in some subset of first-order logic (usually Horn- or definite-clauses). When we refer to a set of clauses, we will usually assume it to be finite. A set of clauses $\mathcal{C} = \{D_1, D_2, \ldots, D_k\}$ will often be used interchangebly with the logical formula $\mathcal{C} = D_1 \wedge D_2 \wedge \cdots \wedge D_k$.

Find: A finite set of clauses (usually in a subset of first-order logic), $H = \{D_1, D_2, \ldots, D_k\}$ s.t.

Strong Posterior Sufficiency: For every $D_j \in H$, $B \cup \{D_j\} \models p_1 \vee p_2 \vee \cdots \vee p_N$
Posterior Satisfiability: $B \cup H \models E^+$
$B \cup H \cup E^- \not\models \square$

Here $\models$ denotes logical consequence and $\square$ denotes a contradiction. MDIE implementations attempt to find the most-probable H, given B and the data E^+, E^-.[2]

The key concept used in Muggleton (1995) is to constrain the identification of the D_j using a *most-specific clause*. The following is adapted from Muggleton (1995).

Remark 1 (Most-Specific Clause) Given background knowledge B and a data-instance e (it does not matter at this point if $e \in E^+$ or $e \in E^-$), any clause D s.t. $B \cup \{D\} \models e$ will satisfy $D \models \overline{B \cup \overline{e}}$. This follows directly from the Deduction Theorem. Let $A = a_1 \wedge a_2 \cdots \wedge a_n$ be the conjunction of ground literals[3] true in all models of $B \cup \overline{e}$. Hence $B \cup \overline{e} \models a_1 \wedge a_2 \wedge \cdots \wedge a_n$. That is, $\overline{a_1 \wedge a_2 \wedge \cdots \wedge a_n} \models \overline{B \cup \overline{e}}$. Let $\bot_B(e)$ denote $\overline{a_1 \wedge a_2 \wedge \cdots \wedge a_n}$. That is, $\bot_B(e)$ is the clause $\neg a_1 \vee \neg a_2 \vee \cdots \vee \neg a_n$. Further, since the a_is are ground, $\bot_B(e)$ is a ground clause.

For any clause D, if $D \models \bot_B(e)$ then $D \models \overline{B \cup \overline{e}}$. Thus any such D satisfies the Weak Posterior Sufficiency condition stated earlier. $\bot_B(e)$ is called the most-specific clause (or "bottom clause") for e, given B.

Thus, bottom clause construction for a data-instance e provides a mechanism for inclusion of all the ground logical consequences given the domain-knowledge B and the instance e.

Example 1 In the following, capitalised letters like X, Y denote variables. Let

B:

$$parent(X,Y) \leftarrow father(X,Y)$$
$$parent(X,Y) \leftarrow mother(X,Y)$$
$$mother(jane, alice) \leftarrow$$

e:

$$gparent(henry, john) \leftarrow father(henry, jane), mother(jane, john)$$

Here "$\leftarrow$" should be read as "if" and the commas (",") as "and". So, the definition for *gparent* is to be read as: "henry is a grandparent of john if henry is the father of jane and jane is the mother of john."

The conjunction A of ground literals true in all models of $B \cup \overline{e}$ is:

[2] Usually, the entailment relation $\models$ is used to identify logical consequences of some set of logical sentences P. That is, what are the e's s.t. $P \models e$? Here, we are given the e's and B, and are asking what is an H s.t. $P = B \cup H$. In this sense, H is said to be the result of inverting entailment (IE).

[3] In theory, the number of ground literals can be infinite. Practical constraints that restrict this number to a finite size are described shortly.

$$\begin{aligned}&\bot_B(e) = \overline{A}:\\ &\quad gparent(henry, john) \leftarrow\\ &\qquad father(henry, jane),\ mother(jane, john),\ mother(jane, alice),\\ &\qquad parent(henry, jane),\ parent(jane, john),\ parent(jane, alice)\end{aligned}$$

The above clause is logically equivalent to the disjunct: *gparent*(*henry*, *john*) $\vee$ $\neg$*father*(*henry*, *jane*) $\vee \cdots \vee$ $\neg$*parent*(*jane*, *alice*). We will also write clause like this as the set: { *gparent*(*henry*, *john*), $\neg$*father*(*henry*, *jane*), …, $\neg$*parent*(*jane*, *alice*) }.

$\bot_B(e)$ thus "extends" the example e to include relations in the background knowledge provided: our interest is in the inclusion of the *parent*/2 relation. The literals in the correct definition of *gparent* is a "generalised" form of subset of the literals in $\bot_B(e)$. Of course, to find the subset and its generalised form efficiently is a different matter, and is the primary concern of ILP systems used to implement MDIE.

It is common to call the non-negated literal in the disjunct (*gparent*(*henry*, *john*)) as the "head" literal, and the negated literals in the disjunct as the "body" literals. In this paper, we will restrict ourselves to $\bot_B(e)$'s that are definite-clauses (clauses with exactly one head literal). This is for practical reasons, and not a requirement of the MDIE formulation of most-specific clauses.

Construction of $\bot_B(e)$ is called a *saturation* step, reflecting the extension of the example by all potentially relevant facts that are derivable using B and the example e. The domain-knowledge can encode significantly more information than simple binary relations (like *parent* above):

Example 2 Suppose data consist of the atom-and-bond structure of molecules that are known to be toxic. Each toxic molecule can be represented by a clausal formula. For example, a toxic molecule m_1 could be represented by the logical formula (here a_1, a_2 are atoms, c denotes carbon, *ar* denotes aromatic, and so on):

$$\begin{aligned}&toxic(m_1) \leftarrow\\ &\quad atom(m_1, a_1, c),\\ &\quad atom(m_1, a_2, c),\\ &\quad \vdots\\ &\quad bond(m_1, a_1, a_2, ar),\\ &\quad bond(m_1, a_2, a_3, ar),\\ &\quad \vdots\end{aligned}$$

The above clause can be read as: molecule m_1 is toxic, if it contains atom a_1 of type carbon, atom a_2 of type carbon, there is an aromatic bond between a_1 and a_2, and so on. We will see later that this is a definite-clause encoding of a graph-based representation of the molecule m_1.

Given background knowledge definitions (for example, of rings and functional groups), $\bot_B(e)$ would extend the logical definition of e with relevant parts of the background knowledge:

$$
\begin{aligned}
toxic(m_1) \leftarrow \\
&atom(m_1, a_1, c), \\
&atom(m_1, a_2, c), \\
&\vdots \\
&bond(m_1, a_1, a_2, ar), \\
&bond(m_1, a_2, a_3, ar), \\
&\vdots \\
&benzene(m_1, [a_1, a_2, a_3, a_4, a_5, a_6]), \\
&benzene(m_1, [a_3, a_4, a_8, a_9, a_{10}, a_{11}]), \\
&\vdots \\
&fused(m_1, [a_1, a_2, a_3, a_4, a_5, a_6], [a_3, a_4, a_8, a_9, a_{10}, a_{11}]), \\
&\vdots \\
&methyl(m_1, [\ldots]), \\
&\vdots
\end{aligned}
$$

As seen from this example, the size of $\bot_B(\cdot)$ can be large. More problematically, for complex domain knowledge, $\bot_B(e)$ may not even be finite. To address this, MDIE introduces the notion of a *depth-bounded* bottom-clause, using *mode* declarations.

3.1 Modes

Practical ILP systems like Progol (Muggleton, 1995) use a *depth-bounded* bottom clause constructed within a mode-language. We first illustrate a simple example of a mode-language specification.

Example 3 A "mode declaration" for an n-arity predicate P (often written as P/n) is one of the following kinds: (a) $modeh(P(a_1, a_2, \ldots, a_n))$; or (b) $modeb(P(a_1, a_2, \ldots, a_n))$. A set of mode-declarations for the predicates in the *gparent* example is: M = { $modeh(gparent(+person, -person))$, $modeb(father(+person, -person))$, $modeb(mother(+person, -person))$, $modeb(parent(+person, -person))$ }.

The *modeh* specifies details about literals that can appear in the head of a clause in the mode-language and the *modeb*'s specify details about literals that can appear in the body of a clause. A "mode declaration" refers to either a *modeh* or *modeb* statement. Based on the mode-language specified in Muggleton (1995), each argument a_i in the mode declarations above is one of: (1) $+person$, denoting that the argument in that literal is an 'input' variable of type *person*.[4] That is, the variable must have appeared either as a $-person$ variable in a literal that appears earlier in the body of the clause or as a $+person$ variable in the head of the clause; (2) $-person$, denoting that the variable in the literal is an 'output' variable of type *person*. If an output variable appears in the head of a clause, it must appear as an output variable of some literal in the body. There are no special constraint on output

[4] Informally, "a variable of type γ" will mean that ground substitutions for the variable are from some set γ. Here, γ is the set $person = \{henry, jane, alice, john, \ldots\}$: that is, *person* is a unary-relation.

variables in body-literals. That is, they can either be a new variable, or any variable (of the same type) that has appeared earlier in the clause. Later we will see how mode-declarations allow the appearance of ground terms.

Example 4 Continuing Example 3, in the following X, Y, Z are variables of type *person*. These clauses are all within the mode language specified in (Muggleton, 1995): (a) $gparent(X, Y) \leftarrow parent(X, Y)$; (b) $gparent(X, Y) \leftarrow parent(X, X)$; (c) $gparent(X, Y) \leftarrow mother(X, Y)$; and (d) $gparent(X, Y) \leftarrow parent(X, Z), parent(Z, Y)$.

But the following clauses are all not within the mode language in Muggleton (1995): (e) $gparent(X, Y) \leftarrow parent(Y, Z)$ (Y does not appear before); (f) $gparent(X, Y) \leftarrow parent(X, Y), parent(Z, Y)$ (Z does not appear before); (g) $gparent(henry, Y) \leftarrow parent(henry, Z), parent(Z, Y)$ (+ arguments have to be variables, not ground terms); and (h) $gparent(X, Y) \leftarrow parent(Z, jane), parent(Z, Y)$ (− arguments have to be variables, not ground terms).

We refer the reader to Muggleton (1995) for more details on the use of modes. Here we confine ourselves to the details necessary for the material in this paper. We first reproduce the notion of a place-number of a term in a literal following (Plotkin, 1972).

Definition 1 (*Term Place-Numbering*) Let $\pi = \langle i_1, \ldots, i_k \rangle$ be a sequence of natural numbers. We say that a term τ is in place-number π of a literal λ iff: (1) $\pi \neq \langle\rangle$; and (2) τ is the term at place-number $\langle i_2, \ldots, i_k \rangle$ in the term at the i_1^{th} argument of λ. τ is at a place-number π in term τ': (1) if $\pi = \langle\rangle$ then $\tau = \tau'$; and (2) if $\pi = \langle i_1, \ldots, i_k \rangle$ then τ' is a term of the form $f(t_1, \ldots, t_m), i_1 \leq m$ and τ is in place-number $\langle i_2, \ldots, i_k \rangle$ in t_{i_1}.

Example 5 (a) In the literal $\lambda = gparent(henry, john)$, the term *henry* occurs in the first argument of λ and *john* occurs in the second argument of λ. The place-numbering of *henry* in λ is $\langle 1 \rangle$ and of *john* in λ is $\langle 2 \rangle$.

(b) As a more complex example, let $\lambda = mem(a, [a, b, c])$ denote the statement that a is a member of the list $[a, b, c]$. The second argument of λ is short-hand for the term $list(a, list(b, list(c, nil)))$ (usually, the function $list/2$ is represented as '.'/2 in the logic-programming literature). Then the term a is a term that occurs in two place-numbers in λ: $\langle 1 \rangle$, and $\langle 2, 1 \rangle$. The term b occurs at place-number $\langle 2, 2, 1 \rangle$ in λ; the term c occurs at place-number $\langle 2, 2, 2, 1 \rangle$ in λ; and the term *nil* occurs at place-number $\langle 2, 2, 2, 2 \rangle$ in λ.

We first present the syntactic aspects constituting a mode-language. The meaning of these elements is deferred to the next section.

Definition 2 (*Mode-Declaration*)

(a) Let Γ be a set of type names. A mode-term is defined recursively as one of: (i) $+\gamma, -\gamma$ or $\#\gamma$ for some $\gamma \in \Gamma$; or (ii) $\phi(mt'_1, mt'_2, \ldots, mt'_j)$, where ϕ is a function symbol of arity

j, and the mt'_ks are mode-terms. We will call mode-terms of type (i) *simple* mode-terms and mode-declarations of type (ii) *structured* mode-terms;[5]

(b) A mode-declaration μ is of the form $modeh(\lambda')$ or $modeb(\lambda')$. Here λ' is a ground-literal of the form $p(mt_1, mt_2, \ldots, mt_n)$ where p is a predicate name with arity n, and the mt_i are mode-terms. We will say μ is a *modeh*-declaration (resp. *modeb*-declaration) for the predicate-symbol p/n.[6] We will also use $ModeLit(\mu)$ to denote λ'.

(c) μ is said to be a mode-declaration for a literal λ iff λ and $ModeLit(\mu)$ have the same predicate symbol and arity.

(d) Let τ be the term at place-number π in μ, We define

$$ModeType(\mu, \pi) = \begin{cases} (+, \gamma) & \text{if } \tau = +\gamma \\ (-, \gamma) & \text{if } \tau = -\gamma \\ (\#, \gamma) & \text{if } \tau = \#\gamma \\ unknown & \text{otherwise} \end{cases}$$

(e) If μ is a mode-declaration for literal λ, $ModeType(\mu, \pi) = (+, \gamma)$ for some place-number π, τ is the term at place π in λ, then we will say τ is an input-term of type γ in λ given μ (or simply τ is an input-term of type γ). Similarly we define output-terms and constant-terms.

3.2 Depth-limited bottom clauses

Returning now to the most-specific clause $\bot_B(e)$ for a data-instance e, given background knowledge B, it is sufficient for our purposes to understand that the input-output specifications in a set of mode-declarations result in a natural notion of the depth at which any term first appears in $\bot_B(e)$ (terms that appear in the head of the clause are at depth 0, terms that appear in literals whose input terms depend only on terms in the head are at depth 1, and so on. A formal definition follows below.) By fixing an upper-bound d on this depth, we can restrict ourselves to a finite-subset of $\bot_B(e)$.[7] This is called the *depth-limited* bottom clause. Given a set of mode-declarations M, we denote this depth-limited clause by $\bot_{B,\mathrm{M},d}(e)$ (or simply $\bot_d(e)$), where d is a (pre-specified) depth-limit. We will refer to the corresponding mode-language as a depth-limited mode-language and denote it by $\mathcal{L}_{\mathrm{M},d}$. We first illustrate this with an example before defining depth formally.

Example 6 Using the modes M in Example 3, we obtain the following most-specific clauses for the *gparent* example (Example 1):

[5] For all experiments in this paper, modes consist only of simple mode-terms.

[6] In general there can be several *modeh* or *modeb*-declarations for a predicate-symbol p/n. If there is exactly one mode-declaration for a predicate symbol p/n, we will say the mode declaration for p/n is determinate.

[7] In fact, additional restrictions are also needed on the number of times a relation can occur at any depth. In implementations like (Muggleton, 1995), this is usually provided as part of the mode declaration.

$$\bot_{B,\mathrm{M},1}(e):$$
$$gparent(henry, john) \leftarrow$$
$$father(henry, jane),$$
$$parent(henry, jane)$$

$$\bot_{B,\mathrm{M},2}(e):$$
$$gparent(henry, john) \leftarrow$$
$$father(henry, jane),$$
$$mother(jane, john),$$
$$mother(jane, alice),$$
$$parent(henry, jane),$$
$$parent(jane, john),$$
$$parent(jane, alice)$$

We now formally define type-definitions and depth for ground-terms.

Definition 3 (*Type Definitions*) Let Γ be a set of types and T be a set of ground-terms. For $\gamma \in \Gamma$ we define a set of ground-terms $T_\gamma = \{\tau_1, \tau_2, \ldots\}$, where $\tau_i \in \mathrm{T}$. We will say a ground-term τ_i is of type γ if $\tau_i \in T_\gamma$, and denote by T_Γ the set $\{T_\gamma : \gamma \in \Gamma\}$. T_Γ will be called a set of type-definitions.

Definition 4 (*Depth of a term*) Let M be a set of modes. Let C be a ground clause. Let λ_i be a literal in C and let τ be an input- or output-term of type γ in λ_i given some $\mu \in \mathrm{M}$. Let Y_τ be the set of all other terms in body literals of C that contain τ as an output-term of type γ. Then,

$$depth(\tau) = \begin{cases} 0 & \text{if } \tau \text{ is an input-term of type } \gamma \\ & \text{in a head literal of } C \\ \min_{\tau' \in Y_\tau} depth(\tau') + 1 & \text{otherwise} \end{cases}$$

Example 7 In the previous example for $C = \bot_{B,\mathrm{M},2}(e)$, $depth(henry) = 0$, $depth(jane) = depth(henry) + 1 = 1$, $depth(john) = depth(jane) + 1 = 2$, and $depth(alice) = depth(jane) + 1 = 2$.

A set of mode-declarations M (see Definition 2), a set of type-definitions T_Γ, and a depth-limit d together define a set of acceptable ground clauses $\mathcal{L}_{T_\Gamma,\mathrm{M},d}$. Informally, $\mathcal{L}_{T_\Gamma,\mathrm{M},d}$ consists of ground clauses in which: (a) all terms are correctly typed; (b) all input terms in a body literal have appeared as output terms in previous body literals or as input terms in any head literal; and (c) all output terms in any head literal appear as output terms in some body literals. In this paper, we will mainly be interested in definite-clauses (that is, $m = 1$ in the definition that follows).

Definition 5 ($\lambda\mu$-*Sequence*) Assume a set of type-definitions T_Γ, modes M, and a depth-limit d. Let $C = \{l_1, \ldots, l_m, \neg l_{m+1}, \ldots, \neg l_k\}$ be a clause with k ground literals. Then $\langle(\lambda_1, \mu_1), (\lambda_2, \mu_2), \ldots, (\lambda_k, \mu_k)\rangle$ is said to be a $\lambda\mu$-sequence for C iff it satisfies the following constraints:

(a) (i) The λ's are all distinct and (ii) For $j = 1 \ldots k$, μ_j is a mode-declaration for λ_j; (iii) For $j = 1 \ldots m$, $\lambda_j = l_j$ and $\mu_j = modeh(\cdot)$; (iv) For $j = (m+1) \ldots k$, $\lambda_j = l_i$ where $\neg l_i \in C$, and $\mu_j = modeb(\cdot)$

(b) If τ is an input-term of type γ in λ_j given μ_j, then:

(i) $\tau \in T_\gamma$; and
(ii) if $j > m$:

- There is an input-term τ of type γ in one of λ_1,...,λ_m given μ_1,...,μ_m; or
- There is an output-term τ of type γ in λ_i ($m<i<j$) given μ_i

(c) If τ is an output-term of type γ in λ_j given μ_j, then

(i) $\tau \in T_\gamma$; and
(ii) if $j \leq m$:

- τ is an output-term of type λ for some λ_i ($m < i \leq k$ given μ_i)

(d) If τ is a constant-term of type γ in λ_j given μ_j then $\tau \in T_\gamma$
(e) There is no term τ at any place π in any λ_j s.t. the $depth(\tau) > d$.

Definition 6 (*Mode-Language*) Assume a set of type-definitions T_Γ, modes M, and a depth-limit d. The mode-language $\mathcal{L}_{T_\Gamma,\mathrm{M},d}$ for T_Γ, M, d is $\{C$: either $C = \emptyset$ or there exists a $\lambda\mu$-sequence for $C\}$.

Example 8 Let M be the set of modes $\{\mu_1, \mu_2, \mu_3, \mu_4, \mu_5, \mu_6\}$ where $\mu_1 = modeh(p(+int))$, $\mu_2 = modeh(p(+real))$, $\mu_3 = modeb(q(+int))$, $\mu_4 = modeb(\ q(+real))$, $\mu_5 = modeb$, $(r(+real))$. Let the depth-limit $d = 1$. Let C be a ground definite-clause $p(1) \leftarrow q(1), r(1)$. That is, $C = \{p(1), \neg q(1), \neg r(1)\}$. Let: $\lambda_1 = p(1), \lambda_2 = q(1), \lambda_3 = r(1)$. Then, C is in $\mathcal{L}_{\mathrm{M},d}$. The $\lambda\mu$-sequences for C are: $\langle(\lambda_1, \mu_1), (\lambda_2, \mu_3), (\lambda_3, \mu_5)\rangle$; $\langle(\lambda_1, \mu_1), (\lambda_3, \mu_5), (\lambda_2, \mu_3)\rangle$; $\langle(\lambda_1, \mu_2), (\lambda_2, \mu_4), (\lambda_3, \mu_6)\rangle$; $\langle(\lambda_1, \mu_2), (\lambda_3, \mu_6), (\lambda_2, \mu_4)\rangle$;

We note that Def. 6 does not allow the following to be $\lambda\mu$-sequences: $\langle(\lambda_1, \mu_1), (\lambda_2, \mu_4), (\lambda_3, \mu_5)\rangle$, $\langle(\lambda_1, \mu_2), (\lambda_2, \mu_4), (\lambda_3, \mu_5)\rangle$, since a 1 of type *int* is treated as being different to a 1 of type *real*.

We note that although the meanings of +, − and # are the same here as in (Muggleton, 1995), clauses in $\mathcal{L}_{T_\Gamma,M,d}$ here are restricted to being ground (in (Muggleton, 1995), clauses are required to have variables in + and − places of literals).

4 BotGNNs

In this section, we describe a method to translate the depth-limited most-specific clauses of the previous section ($\bot_{B,\mathrm{M},d}(\cdot)$'s) into a form that can be used by standard variants of GNNs. We illustrate the procedure first with an example.

Example 9 Consider $\bot_{B,\mathrm{M},2}(e)$ in Example 6. The tabulation below shows the literals in $\bot_{B,\mathrm{M},2}(e)$ and matching modes

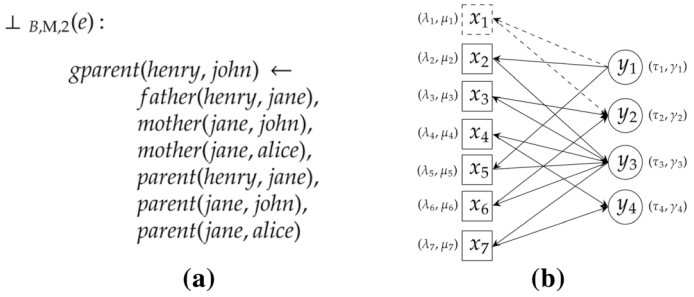


Fig. 2 For the *gparent* example: **a** depth-limited bottom-clause $\perp_{B,\mathrm{M},2}(e)$; and **b** the corresponding clause-graph where the vertex-labels (λ, μ)s and (τ, γ)s are as provided in the preceding tables. The "dashed" square-box and the "dashed" arrow are shown to indicate the vertex specifying the head of the clause. The subscripts used in the labels correspond to the s.no. in the tables, for example, (λ_3, μ_3) refers to the third-row in the first table in this example; and, similarly, (τ_4, γ_4) refers to the fourth row in the second table

S.No.	Literal (λ)	Mode (μ)
1	*gparent*(*henry*, *john*)	*modeh*(*gparent*(+*person*, −*person*))
2	*father*(*henry*, *jane*)	*modeb*(*father*(+*person*, −*person*))
3	*mother*(*jane*, *john*)	*modeb*(*mother*(+*person*, −*person*))
4	*mother*(*jane*, *alice*)	*modeb*(*mother*(+*person*, −*person*))
5	*parent*(*henry*, *jane*)	*modeb*(*parent*(+*person*, −*person*))
6	*parent*(*jane*, *john*)	*modeb*(*parent*(+*person*, −*person*))
7	*parent*(*jane*, *alice*)	*modeb*(*parent*(+*person*, −*person*))

The table below shows the ground-terms (τ's) in literals appearing in $\perp_{B,\mathrm{M},2}(e)$ and their types (γ's), obtained from the corresponding term-place number in the matching mode:

S.No.	Term (τ)	Type (γ)
1	*henry*	*person*
2	*john*	*person*
3	*jane*	*person*
4	*alice*	*person*

The information in these tables can be represented as a directed bipartite graph as shown in Fig. 2. The square-shaped vertices represent (λ, μ) pairs in the first table, and the round-shaped vertices represent (τ, γ) pairs in the second table. Arcs from a (λ, μ) (square-) vertex to a (τ, γ) (round-) vertex indicates term τ is designated by mode μ as an output or constant term ($-$ or #) of type γ in literal λ. Conversely, an arc from an (τ, γ) vertex to an (λ, μ) vertex indicates that term τ is designated by mode μ as an input term (+) of type γ in literal λ.

The structure in Fig. 2 is called a bottom-graph in this paper. BotGNNs are GNN models constructed from graphs based on such clause-graphs. We first clarify some details needed for the construction of clause-graphs.

4.1 Notations and assumptions

Sets: We use the following notations:

a. Set $\mathcal{E}$ to define a set of relational data-instances[8];
b. Sets P, F, K to denote predicate-symbols, function-symbols, and constant-symbols, respectively;
c. Λ to denote the set of all positive ground-literals that can be constructed using P, F, K; and T to denote the set of all ground-terms that can be constructed using F, K;[9]
d. Λ_C to denote the set of all literals in a clause C;
e. B to denote the set of predicate-definitions constituting background knowledge;
f. M to denote the set of modes for the predicate-symbols in P;
g. Let Γ' to denote the set of type-names used by modes in M. In addition, we assume a special type-name $\mathbb{R}$ to denote a numeric type. We denote by Γ the set $\Gamma' \cup \{\#t : t \in \Gamma'$ s.t. $\#t$ occurs in some mode $\mu \in \text{M}\}$;
h. LM to denote the set $\{(\lambda, \mu) : \lambda \in \Lambda, \mu \in \text{M}, \mu$ is a mode-declaration for $\lambda\}$; and ET to denote the set $\{(\tau, \gamma) : \tau \in \text{T}, \gamma \in \Gamma, \tau$ is of type $\gamma\}$;
i. Xs to denote the set $\{x_1, \ldots, x_{|LM|}\}$ and Ys to denote the set $\{y_1, \ldots, y_{|ET|}\}$;
j. $\mathcal{B}$ to denote the set of bipartite graphs[10] of the form (X, Y, E) where $X \subseteq Xs$, $Y \subseteq Ys$, and $E \subseteq (Xs \times Ys) \cup (Ys \times Xs)$;
k. $\mathcal{G}$ to denote the set of labelled bipartite graphs $((X, Y, E), \psi)$ where $(X, Y, E) \in \mathcal{B}$ and $\psi : (Xs \cup Ys) \rightarrow (LM \cup ET)$.
l. We will use $CG_{\top}$ to denote the special graph $((\emptyset, \emptyset, \emptyset), \emptyset) \in \mathcal{G}$.

Functions: We assume bijections $h_x : LM \rightarrow Xs$; and $h_y : ET \rightarrow Ys$;
Implementation: We will assume the following implementation details:

a. The elements of Γ are assumed to be unary predicate symbols, and the type-definitions T_Γ in Definition 3 will be implemented as predicate-definitions in B. That is, if a ground-term τ is of type $\gamma \in \Gamma$ (that is $\tau \in T_\gamma$ in Definition 3) then $\gamma(\tau) \in B$. We will therefore refer to the mode-language $\mathcal{L}_{T_\Gamma,\text{M},d}$ in Definition 6 as $\mathcal{L}_{B,\text{M},d}$;
b. An MDIE implementation that, given B, M, d, ensures for any ground definite-clause e returns a unique ground definite-clause $\bot_{B,\text{M},d}(e) \in \mathcal{L}_{B,\text{M},d}$ if it exists or $\emptyset$ otherwise. In addition, if $\bot_{B,\text{M},d}(e) \in \mathcal{L}_{B,\text{M},d}$, we assume the MDIE implementation has been extended to return at least one matching $\lambda\mu$-sequence for $\bot_{B,\text{M},d}$.

4.2 Construction of bottom-graphs

We now define the graph-structures or simply, the graphs constructed from the depth-limited bottom-clauses.

[8] In this paper, this set consists of definite clauses.

[9] A *term* is defined recursively as a constant from K, a variable, or a function symbol from F applied to term. A ground term is a term without any variables.

[10] A directed graph $G = (V, E)$ is called bipartite if there is a 2-partition of V into sets X, Y, s.t. there are no vertices $a, b \in X$ (resp. Y) s.t. $(a, b) \in E$. We will sometimes denote such a bipartite graph by (X, Y, E), where it is understood that $V = X \cup Y$.

Definition 7 (*Literals Set*) Given background knowledge B, a set of modes M, a depth-limit d, let C be a clause in $\mathcal{L}_{B,\mathrm{M},d}$. We define $Lits_{B,\mathrm{M},d}(C)$, or simply $Lits(C)$ as follows:

(i) If $C = \emptyset$ then $Lits(C) = \emptyset$;
(ii) If $C \neq \emptyset$, let $\mathcal{LM}$ be the set of all $\lambda\mu$-sequences for C. Then $Lits(C) = \{(\lambda_i, \mu_i) : S \in \mathcal{LM}$ and (λ_i, μ_i) is in sequence $S\}$

The definition for $Lits(\cdot)$ requires all $\lambda\mu$-sequences to ensure that *Lits* is well-defined. In practice, we restrict ourselves to the $\lambda\mu$-sequences identified by the MDIE implementation. If these are a subset of all $\lambda\mu$-sequences, then the resulting clause-graph will be "more general" than that obtained with all $\lambda\mu$-sequences (see Appendix A).

Example 10 We revisit the *gparent* example. Let M $= \{\mu_1, \mu_2, \mu_3, \mu_4\}$, where $\mu_1 = modeh(gparent(+person, -person))$; $\mu_2 = modeb(father(+person, -person))$; $\mu_3 = modeb(mother(+person, -person))$; $\mu_4 = modeb(parent(+person, -person))$. Let background knowledge B contain the type-definitions: $person(henry)$, $person(john)$, $person(jane)$, $person(alice)$; and let depth-bound $d = 2$, Let $C = \bot_{B,\mathrm{M},d}(e)$ as in Example 6.

1. Here $C = \{gparent(henry, john)$, $\neg father(henry, jane)$, $\neg mother(jane, john)$, $\neg mother(jane, alice)$, $\neg parent(henry, jane)$, $\neg parent(jane, john)$, $\neg parent(jane, alice)\}$.
2. $\Lambda_C = \{ \lambda_1, \lambda_2, \ldots, \lambda_7\}$ where: $\lambda_1 = gparent(henry, john)$, $\lambda_2 = father(henry, jane)$, $\lambda_3 = mother(jane, john)$, $\lambda_4 = mother(jane, alice)$, $\lambda_5 = parent(henry, jane)$, $\lambda_6 = parent(jane, john)$, $\lambda_7 = parent(jane, alice)$
3. $C \in \mathcal{L}_{B,\mathrm{M},d}$ because $S = \langle(\lambda_1, \mu_1), (\lambda_2, \mu_2), (\lambda_3, \mu_3), (\lambda_4, \mu_3), (\lambda_5, \mu_4), (\lambda_6, \mu_4), (\lambda_7, \mu_4)\rangle$ is a $\lambda\mu$-sequence for C. Some other permutations of S will also be $\lambda\mu$-sequences. The reader can verify that the terms in λ-components of S are correctly typed; input terms in the body literals appear after corresponding output terms in body-literals earlier in the λ-components of S, or as input-terms in λ_1; the output-term in λ_1 appears as an output-term in some λ later in the sequence S.
4. Then $Lits(C) = \{(\lambda_1, \mu_1), (\lambda_2, \mu_2), (\lambda_3, \mu_3), (\lambda_4, \mu_3), (\lambda_5, \mu_4), (\lambda_6, \mu_4), (\lambda_7, \mu_4)\}$.

Definition 8 (*Terms Set*) Given background knowledge B, a set of modes M, a depth-limit d, let $C \in \mathcal{L}_{B,\mathrm{M},d}$. We define $Terms_{B,\mathrm{M},d}(C)$, or simply $Terms(C)$ as follows.

If $Lits(C) = \emptyset$, then $Terms(C) = \emptyset$. Otherwise, for any pair $(\lambda, \mu) \in Lits(C)$, let $Ts((\lambda, \mu)) = \{(\lambda, \mu, \pi) : \pi$ is a place-number s.t. $ModeType(\mu, \pi) = (\cdot, \gamma)$ for some $\gamma \in \Gamma\}$. Then $Terms(C) = \bigcup_{x \in Lits(C)} Ts(x)$.

Example 11 In Example 10, $Lits(C) = \{(\lambda_1, \mu_1)$, (λ_1, μ_1), ..., $(\lambda_7, \mu_4)\}$. Therefore, $Terms(C) = \{(\lambda_1, \mu_1, \langle 1\rangle)$, $(\lambda_1, \mu_1, \langle 2\rangle)$, $(\lambda_2, \mu_2, \langle 1\rangle)$, $(\lambda_2, \mu_2, \langle 2\rangle)$, $...(\lambda_7, \mu_4, \langle 1\rangle)$, $(\lambda_7, \mu_4, \langle 2\rangle)\}$.

Definition 9 (*Clause-Graphs*) Given background knowledge B, a set of modes M, and a depth-limit d, we define a function $ClauseToGraph : \mathcal{L}_{B,\mathrm{M},d} \rightarrow \mathcal{G}$ as follows.

If $C = \emptyset$ then $ClauseToGraph(C) = CG_{\top}$ (see Sect. 4.1). Otherwise, $ClauseToGraph(C) = ((X, Y, E), \psi) \in \mathcal{G}$ where:

(a) $X = \{x_i : (\lambda, \mu) \in Lits(C), x_i = h_x((\lambda, \mu))\}$;

(b) $Y = \{y_j : (\lambda, \mu, \pi) \in Terms(C)\ ,\quad TermType((\lambda, \mu, \pi)) = (\tau, \gamma)\ ,\quad ModeType(\mu, \pi) \in \{(+, \gamma), (-, \gamma)\}, y_j = h_y((\tau, \gamma))\} \cup \{y_j : (\lambda, \mu, \pi) \in Terms(C), TermType((\lambda, \mu, \pi)) = (\tau, \gamma)\ TermType((\lambda, \mu, \pi)) = (\tau, \gamma), ModeType(\mu, \pi) = (\#, \gamma), y_j = h_y((\tau, \#\gamma))\}$;

(c) $E = E_{in} \cup E_{out}$, where:

$$E_{in} = \{(y_j, x_i) : (\lambda, \mu, \pi) \in Terms(C), x_i = h_x((\lambda, \mu)), (\tau, \gamma) = TermType((\lambda, \mu, \pi)), y_j = h_y((\tau, \gamma)), ModeType(\mu, \pi) = (+, \gamma)\},$$

$$\text{and, } E_{out} = \{(x_i, y_j) : (\lambda, \mu, \pi) \in Terms(C), x_i = h_x((\lambda, \mu)), (\tau, \gamma) = TermType((\lambda, \mu, \pi)), y_j = h_y((\tau, \gamma)), ModeType(\mu, \pi) \in \{(-, \gamma), (\#, \gamma)\}\}$$

and ψ is a vertex-labelling function defined as follows:

(d) For $v \in X$, $\psi(v) = h_x^{-1}(v)$;

(e) For $v \in Y$, $\psi(v) = h_y^{-1}(v)$

In Appendix A, we show $ClauseToGraph(\cdot)$ is an injective function.

Example 12 We continue Example 11. Recall $Terms(C) = \{\ (\lambda_1, \mu_1, \langle 1\rangle), (\lambda_1, \mu_1, \langle 2\rangle), (\lambda_2, \mu_2, \langle 1\rangle), ..., (\lambda_7, \mu_4, \langle 2\rangle)\}$. Then, in Definition 9, $TermType((\lambda_1, \mu_1, \langle 1\rangle)) = (henry, person)$, $TermType((\lambda_1, \mu_1, \langle 2\rangle)) = (john, person)$, $TermType(\ (\lambda_2, \mu_2, \langle 1\rangle)) = (henry, person)$, ..., $TermType((\lambda_7, \mu_4, \langle 2\rangle)) = (alice, person)$. Then $ClauseToGraph(C)$ is as follows:

- $G = (X, Y, E)$ where:
  - $X = \{x_1, x_2, \ldots, x_7\}$, where: $x_1 = h_x((\lambda_1, \mu_1))$; $x_2 = h_x((\lambda_2, \mu_2))$; $...x_7 = h_x((\lambda_7, \mu_4))$
  - $Y = \{y_1, y_2, y_3, y_4\}$ where: $y_1 = h_y((henry, person))$; $y_2 = h_y((john, person))$; $y_3 = h_y((jane, person))$; $y_4 = h_y((alice, person))$
  - $E = E_{in} \cup E_{out}$, where

    $E_{in} = \{(y_1, x_1), (y_1, x_2), (y_1, x_5), (y_3, x_3), (y_3, x_4), (y_3, x_6), (y_3, x_7)\}$
    $E_{out} = \{(x_1, y_2), (x_2, y_3), (x_3, y_2), (x_4, y_4), (x_5, y_3), (x_6, y_2), (x_7, y_4)\}$
- The vertex-labelling ψ is s.t. $\psi(x_1) = (\lambda_1, \mu_1)$; $\psi(x_2) = (\lambda_2, \mu_2)$; $\psi(x_3) = (\lambda_3, \mu_3)$; $\psi(x_4) = (\lambda_4, \mu_3)$; $\psi(x_5) = (\lambda_5, \mu_4)$; $\psi(x_6) = (\lambda_6, \mu_4)$; $\psi(x_7) = (\lambda_7, \mu_4)$; $\psi(y_1) = (henry, person)$; $\psi(y_2) = (john, person)$; $\psi(y_3) = (jane, person)$; $\psi(y_4) = (alice, person)$.

The reader can compare this to the graph shown diagrammatically in Fig. 2.

Example 13 Examples 10–12 do not illustrate what happens when we have multiple matching mode-declarations. To illustrate this we repeat the exercise with Example 8 (for consistency, we now use $\mathbb{R}$ instead of *real*) In that example, $\mathrm{M} = \{\mu_1, \mu_2, \mu_3, \mu_4, \mu_5, \mu_6\}$ where $\mu_1 = modeh(p(+int))$, $\mu_2 = modeh(p(+\mathbb{R}))$, $\mu_3 = modeb(q(+int))$, $\mu_4 = modeb(q(+\mathbb{R}))$, $\mu_5 = modeb(r(+int))$, $\mu_6 = modeb(r(+\mathbb{R}))$. Let the depth-limit $d = 1$.

1. Here $C = \{\ p(1), \neg q(1), \neg r(1)\ \}$;
2. $\Lambda_C = \{\lambda_1, \lambda_2, \lambda_3\}$, where $\lambda_1 = p(1)$, $\lambda_2 = q(1)$, $\lambda_3 = r(1)$.
3. $C \in \mathcal{L}_{B,\mathrm{M},d}$ since there is at least one $\lambda\mu$-sequence for C (in fact, there are 4 matching $\lambda\mu$ -sequences: see Example 8).

4. $Lits(C) = \{(\lambda_1, \mu_1), (\lambda_2, \mu_3), (\lambda_3, \mu_5), (\lambda_1, \mu_2), (\lambda_2, \mu_4), (\lambda_3, \mu_6)\}$.
5. We note that the term 1 is at place-number $\langle 1 \rangle$ in all the three literals.
6. Then $Terms(C) = \{(\lambda_1, \mu_1, \langle 1 \rangle), (\lambda_2, \mu_3, \langle 1 \rangle), ..., (\lambda_3, \mu_6, \langle 1 \rangle)\}$
7. Then, in Definition 9, $TermType((\lambda_1, \mu_1, \langle 1 \rangle)) = (1, int)$, $TermType((\lambda_2, \mu_3, \langle 1 \rangle)) = (1, int)$, ..., $TermType((\lambda_3, \mu_6, \langle 1 \rangle)) = (1, \mathbb{R})$.

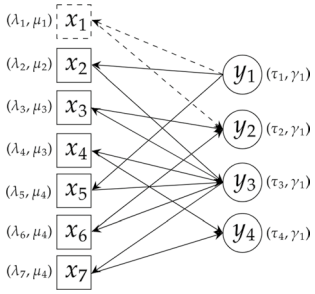
The reader can verify that $ClauseToGraph(C) = (G, \cdot)$ where $G = (X, Y, E)$ s.t.

- $X = \{x_1, x_2, \ldots, x_6\}$, where $x_1 = h_x((\lambda_1, \mu_1))$, $x_2 = h_x((\lambda_2, \mu_3))$,..., $x_6 = h_x((\lambda_3, \mu_6))$
- $Y = \{y_1, y_2\}$, where $y_1 = h_y((1, int))$ and $y_2 = h_y((1, \mathbb{R}))$
- $E = \{(y_1, x_1), (y_1, x_2), (y_1, x_3), ((y_2, x_4), (y_2, x_5), (y_2, x_6)\}$

It is now straightforward to define graphs from most-specific clauses.

Definition 10 (*Bottom-Graphs*) Given a data instance $e \in \mathcal{E}$ and B, M, d as before, let $\bot_{B,\mathrm{M},d}(e)$ be the (depth-bounded) most-specific ground definite-clause for e. We define $BotGraph_{B,\mathrm{M},d}(e) : \mathcal{E} \to \mathcal{G}$, or simply $BotGraph(e)$ as follows: $BotGraph(e) = ClauseToGraph(\bot_{B,\mathrm{M},d}(e))$.

Example 14 For our *gparent*/2 example described through out this paper, the bottom-graph for the most-specific clause with $d = 2$ is written as: $BotGraph(e) = ClausetoGraph(\bot_{B,\mu,2}(e))$, which is shown in the diagram below (the "dashed" square-box and the "dashed" arrow are shown to indicate the vertex specifying the head of the clause):



The vertex-labelling in the above graph is as obtained in Example 12, where γ_1 denotes the type-name *person*, $\tau_1, \ldots, \tau_4$ denote the terms *henry*, *john*, *jane*, *alice* respectively. The reader can verify that the diagram above is consistent with the bottom-graph shown in Fig. 2.

Some properties of clause-graphs are in Appendix A. The bottom-graphs defined here are not immediately suitable for GNNs for the task of graph-classification. Some graph-transformations are needed before providing them as input to a GNN. We describe these transformations next.

4.3 Transformations for graph classification by a GNN

We now describe functions used to transform bottom-graphs into a form suitable for the GNN implementations we consider in this paper. The definite-clause representation of graphs that we use (an example follows below) contains all the information about the graph in the antecedent of the definite-clause. The following function extracts the corresponding parts of the bottom-graph.

Definition 11 (*Antecedent-Graphs*) We define function $Antecedent : \mathcal{G} \rightarrow \mathcal{G}$ as follows. Let $(G, \psi) \in \mathcal{G}$, where $G = (X, Y, E)$ is a directed bipartite graph. Let $X_h = \{x : x \in X, \psi(x) = (\lambda, \mu), \mu = modeh(\cdot)\}$. We define (G', ψ') where $G' = (X', Y', E')$ and

- $X' = X - X_h$
- $Y' = \{y : y \in Y, \exists x \in X' \text{ s.t. } (x, y) \in E \text{ or } (y, x) \in E\}$
- $E' = E - \{(v_i, v_j) : v_i \in X_h\} - \{(v_j, v_i) : v_i \in X_h\}$

and, $\psi'(v_i) = \psi(v_i)$ for all $v_i \in X' \cup Y'$. Then $Antecedent((G, \psi)) = (G', \psi')$.

Most GNN implementations, including those used in this paper, require graphs to be undirected (Hamilton, 2020). Furthermore, an undirected graph representation allows an easy exchange of messages across multiple relations (the X-nodes) resulting in unfolding their internal dependencies. We define a function that converts directed clause-graphs to undirected clause-graphs.

Definition 12 (*Undirected Clause-Graphs*) We define a function $UGraph : \mathcal{G} \rightarrow \mathcal{G}$ as follows. Let $(G, \psi) \in \mathcal{G}$, where $G = (X, Y, E)$ is a directed bipartite graph. We define (G', ψ'), where $G' = (X', Y', E')$ and

- $X' = X$
- $Y' = Y$
- $E' = E \cup \{(v_j, v_i) : (v_i, v_j) \in E\}$

and $\psi'(v_i) = \psi(v_i)$ for all $v_i \in X' \cup Y'$. Then $UGraph((G, \psi)) = (G', \psi')$.

In fact, graphs for GNNs are not actually in $\mathcal{G}$. GNN implementations usually require vertices in a graph to be labelled with numeric feature-vectors. This requires a modification of the vertex-labelling to be a function from vertices to real-vectors of some finite length. The final transformation converts the vertex-labelling of a graph in $\mathcal{G}$ into a suitable form.

Definition 13 (*Vectorise*) Let $(G, \psi) \in \mathcal{G}$, where $G = (X, Y, E)$. Assume we are given a set of modes M. Let $\Gamma_{\#}$ be the set of all type-names $\gamma \in \Gamma - \{\mathbb{R}, \#\mathbb{R}\}$ such that $\#\gamma$ in some mode $\mu \in$ M. Let $\mathrm{T}_{\#} \subseteq$ T be the set of ground-terms of types in $\Gamma_{\#}$.[11]

Let us define the following four functions from $X \cup Y$ to the set of all real vectors of finite length. For $v \in X \cup Y$:

[11] That is, $\Gamma_{\#}$ is the set of all #-ed, non-numeric type-names in M. and $\mathrm{T}_{\#}$ is the set of all ground-terms of #-ed non-numeric types.

$$f_\rho(v) = \begin{cases} onehot(P, r) & \text{if } v \in X, h(v) = (\lambda, \cdot) \text{ and } predsym(\lambda) = r \\ \mathbf{0}^{|P|} & \text{otherwise} \end{cases}$$

$$f_\gamma(v) = \begin{cases} onehot(\Gamma, \gamma) & \text{if } v \in Y \text{ and } h(v) = (\tau, \gamma) \\ \mathbf{0}^{|\Gamma|} & \text{otherwise} \end{cases}$$

$$f_\tau(v) = \begin{cases} onehot(\mathrm{T}_\#, \tau) & \text{if } v \in Y \text{ and } h(v) = (\tau, \#\gamma) \text{ and } \gamma \notin \mathbb{R} \\ \mathbf{0}^{|\mathrm{T}_\#|} & \text{otherwise} \end{cases}$$

$$f_\mathbb{R}(v) = \begin{cases} [\tau] & \text{if } v \in Y \text{ and } h(v) = (\tau, \#\mathbb{R}) \\ \mathbf{0}^1 & \text{otherwise} \end{cases}$$

where $\mathbf{0}^d$ denotes the zero-vector of length d; $predsym(l)$ is a function that returns the name and arity of literal l; and $onehot(S, x)$ denotes a one-hot vector encoding of $x \in S$.[12]

Let *Vectorise* be a function defined on $\mathcal{G}$ as follows: $Vectorise((G, \psi)) = (G, \psi')$ where $\psi'(v) = f_\rho(v) \oplus f_\gamma(v) \oplus f_\tau(v) \oplus f_\mathbb{R}(v)$ for each $v \in X \cup Y$. Here $\oplus$ denotes vector concatenation.

We note that the vectors in the vertex-labelling from *Vectorise* should not be confused with the vector obtained using the *Vec* function employed within a GNN (see Fig. 1) in Sect. 2. The purpose of that function is to obtain a low-dimensional real-valued vector representation for an entire graph (usually for problems of graph-classification).

Example 15 Recall the most-specific clause for the *gparent*(*henry*, *john*) in Example 1: *gparent*(*henry*, *john*) ← *father*(*henry*, *jane*), *mother*(*jane*, *john*), *mother*(*jane*, *alice*), *parent*(*henry*, *jane*), *parent*(*jane*, *john*), *parent*(*jane*, *alice*). The clause-graph and corresponding antecedent-graph are shown below.

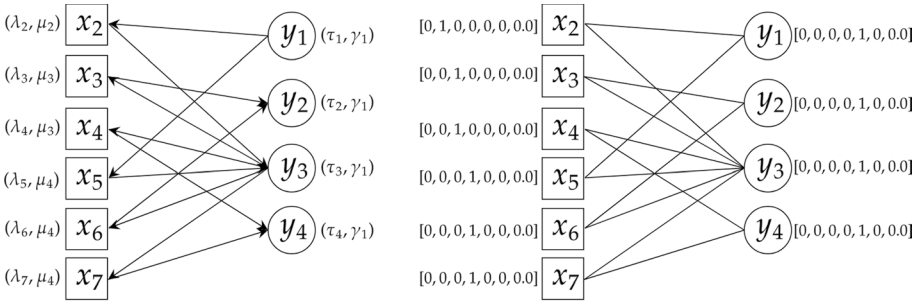
Assume the following sets: $P = \{gparent/2, father/2, mother/2, parent/2\}$, $\Gamma = \{person\}$, $\Gamma_\# = \emptyset$, $\mathrm{T}_\# = \emptyset$.

Additionally, since the mode-language in Example 1 does not have any #'ed arguments, $\mathrm{T}_\# = \emptyset$. So: f_ρ is a 4-dimensional (one-hot encoded) vector (since $|P| = 4$); f_γ is a 1-dimensional vector (since $|Gamma| = 1$); f_τ is a 1-dimensional vector containing 0 (since $|\mathrm{T}_\#| = 0$); and $f_\mathbb{R}$ is a 1-dimensional vector containing 0 (since there are no #'ed numeric terms)). A full tabulation of the vectors involved is provided below, along with the new vertex-labelling that results. In the table, the vertex labels are as obtained in Example 12; γ_1 is used to denote the type *person*.

v	$\psi(v)$	$f_\rho(v)^\top$	$f_\gamma(v)^\top$	$f_\tau(v)^\top$	$f_\mathbb{R}(v)^\top$	$\psi'(v)^\top$
x_2	(λ_2, μ_2)	[0, 1, 0, 0]	[0]	[0]	[0.0]	[0, 1, 0, 0, 0, 0, 0.0]
x_3	(λ_3, μ_3)	[0, 0, 1, 0]	[0]	[0]	[0.0]	[0, 0, 1, 0, 0, 0, 0.0]
x_4	(λ_4, μ_3)	[0, 0, 1, 0]	[0]	[0]	[0.0]	[0, 0, 1, 0, 0, 0, 0.0]
x_5	(λ_5, μ_4)	[0, 0, 0, 1]	[0]	[0]	[0.0]	[0, 0, 0, 1, 0, 0, 0.0]
x_6	(λ_6, μ_4)	[0, 0, 0, 1]	[0]	[0]	[0.0]	[0, 0, 0, 1, 0, 0, 0.0]
x_7	(λ_7, μ_4)	[0, 0, 0, 1]	[0]	[0]	[0.0]	[0, 0, 0, 1, 0, 0, 0.0]
y_1	(τ_1, γ_1)	[0, 0, 0, 0]	[1]	[0]	[0.0]	[0, 0, 0, 0, 1, 0, 0.0]
y_2	(τ_2, γ_1)	[0, 0, 0, 0]	[1]	[0]	[0.0]	[0, 0, 0, 0, 1, 0, 0.0]
y_3	(τ_3, γ_1)	[0, 0, 0, 0]	[1]	[0]	[0.0]	[0, 0, 0, 0, 1, 0, 0.0]
y_4	(τ_4, γ_1)	[0, 0, 0, 0]	[1]	[0]	[0.0]	[0, 0, 0, 0, 1, 0, 0.0]

[12] A one-hot vector encoding of an element x in a set S assumes a 1-1 mapping N from elements of S to $\{1, \ldots, |S|\}$. If $x \in S$ and $onehot(S, x) = \mathbf{v}$ then $\mathbf{v}$ is a vector of dimension $|S|$ s.t. $N(x)$'th entry in $\mathbf{v}$ is 1 and all other entries in $\mathbf{v}$ are 0.

The following figures show: (a) the antecedent graph and (b) the vectorised, undirected, antecedent graph for the *gparent* example. We call the structure in (b) as a BotGNNGraph, the definition of which is provided later.



(a) Antecedent-graph **(b)** Vectorised, undirected, antecedent-graph

The example above does not have any #-ed arguments in the modes M. In the following example, we consider modes that have #-ed arguments (of types: $\mathbb{R}$ and not $\mathbb{R}$) and repeat the same exercise: starting with the construction of the bottom-graph. Then we show how the function *Vectorise* results in a vectorised graph suitable for a GNN.

Example 16 Let M be the set of modes $\{\mu_1, \mu_2, \mu_3\}$ where $\mu_1 = modeh(p(+\mathbb{R}))$, $\mu_2 = modeb(q(+\mathbb{R}, \#colour))$, $\mu_3 = modeb(r(\#colour, \#\mathbb{R}))$. Let the depth-limit $d = 1$ and that the background knowledge contains the type-definitions $colour(white)$ and $colour(black)$. Let C be a ground definite-clause $p(1.0) \leftarrow q(1.0, white), r(white, 1.0)$. The following are obtained based on the definitions:

- $C = \{p(1.0), \neg q(1.0, white), \neg r(white, 1.0)\}$.
- $\Lambda_C = \{\lambda_1, \lambda_2, \lambda_3\}$, where $\lambda_1 = p(1.0)$, $\lambda_2 = q(1.0, white)$, $\lambda_3 = r(white, 1.0)$.
- C is in $\mathcal{L}_{B,\mathrm{M},d}$ since there is at least one $\lambda\mu$-sequence for C. Here we have one such sequence: $\langle(\lambda_1, \mu_1), (\lambda_2, \mu_2), (\lambda_3, \mu_3)\rangle$
- $Lits(C) = \{(\lambda_1, \mu_1), (\lambda_2, \mu_2), (\lambda_3, \mu_3)\}$
- $Terms(C) = \{(\lambda_1, \mu_1, \langle 1\rangle), (\lambda_2, \mu_2, \langle 1\rangle), (\lambda_2, \mu_2, \langle 2\rangle), (\lambda_3, \mu_3, \langle 1\rangle)(\lambda_3, \mu_3, \langle 2\rangle)\}$
- $TermType((\lambda_1, \mu_1, \langle 1\rangle)) = (1.0, \mathbb{R})$, $TermType((\lambda_2, \mu_2, \langle 1\rangle)) = (1.0, \mathbb{R})$, $TermType((\lambda_2, \mu_2, \langle 2\rangle)) = (white, \#colour)$, $TermType((\lambda_3, \mu_3, \langle 1\rangle)) = (1.0, \#\mathbb{R})$ $TermType((\lambda_3, \mu_3, \langle 2\rangle)) = (white, \#colour)$

Then, $ClauseToGraph(C) = (G, \psi)$, where $G = (X, Y, E)$ s.t.

- $X = \{x_1, x_2, x_3\}$, where $x_1 = h_x((\lambda_1, \mu_1))$, $x_2 = h_x((\lambda_2, \mu_2))$ and $x_3 = h_x((\lambda_3, \mu_3))$
- $Y = \{y_1, y_2, y_3\}$, where $y_1 = h_y((1.0, \mathbb{R}))$, $y_2 = h_y((white, \#colour))$, $y_3 = h_y((1.0, \#\mathbb{R}))$
- $E = \{(y_1, x_1), (y_1, x_2), (x_2, y_2), (x_3, y_2), (x_3, y_3)\}$

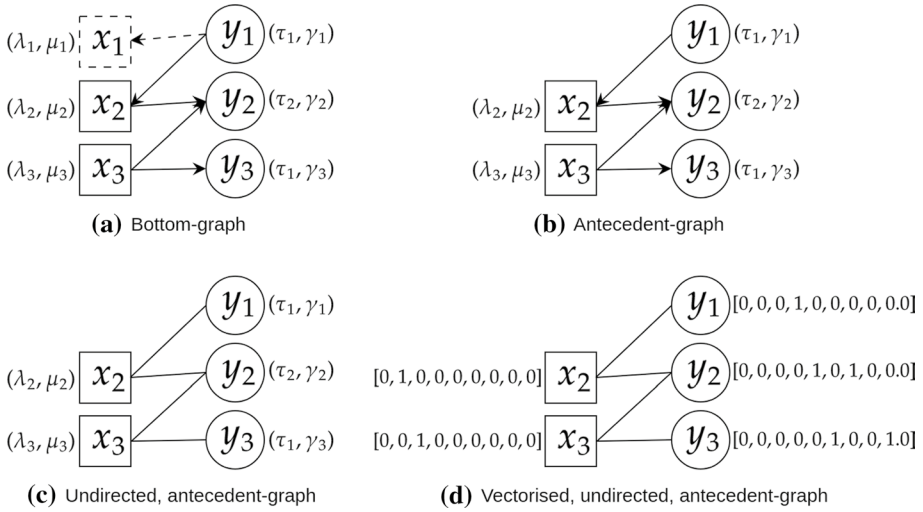
and, the vertex-labelling ψ is as follows: $\psi(x_1) = (\lambda_1, \mu_1)$, $\psi(x_2) = (\lambda_2, \mu_2)$, $\psi(x_3) = (\lambda_3, \mu_3)$, $\psi(y_1) = (1.0, \mathbb{R})$, $\psi(y_2) = (white, \#class)$, $\psi(y_3) = (1.0, \#\mathbb{R})$.

In this example, we assume the following sets: $P = \{p/1, q/2, r/2\}$, $\Gamma = \{\mathbb{R}, \#colour, \#\mathbb{R}\}$, $\Gamma_\# = \{\#colour\}$, $\mathrm{T}_\# = \{white, black\}$.

The graph (G, ψ) constructed above is the bottom-graph for this particular example. The feature-vectors obtained from the functions in *Vectorise* are tabulated below. In the table, $\tau_1 = 1.0$, $\tau_2 = white$, $\gamma_1 = \mathbb{R}$, $\gamma_2 = \#colour$, $\gamma_3 = \#\mathbb{R}$.

v	$\psi(v)$	$f_\rho(v)^\top$	$f_\gamma(v)^\top$	$f_\tau(v)^\top$	$f_{\mathbb{R}}(v)^\top$	$\psi'(v)^\top$
x_2	(λ_2, μ_2)	[0, 1, 0]	[0, 0, 0]	[0, 0]	[0.0]	[0, 1, 0, 0, 0, 0, 0, 0, 0.0]
x_3	(λ_3, μ_3)	[0, 0, 1]	[0, 0, 0]	[0, 0]	[0.0]	[0, 0, 1, 0, 0, 0, 0, 0, 0.0]
y_1	(τ_1, γ_1)	[0, 0, 0]	[1, 0, 0]	[0, 0]	[0.0]	[0, 0, 0, 1, 0, 0, 0, 0, 0.0]
y_2	(τ_2, γ_2)	[0, 0, 0]	[0, 1, 0]	[1, 0]	[0.0]	[0, 0, 0, 0, 1, 0, 1, 0, 0.0]
y_3	(τ_1, γ_3)	[0, 0, 0]	[0, 0, 1]	[0, 0]	[1.0]	[0, 0, 0, 0, 0, 1, 0, 0, 1.0]

The following figure shows how the final vectorised graph is constructed from the bottom-graph (the dotted square-box and the dotted arrow are shown to indicate the vertex specifying the head of the clause C):



(a) Bottom-graph **(b)** Antecedent-graph

(c) Undirected, antecedent-graph **(d)** Vectorised, undirected, antecedent-graph

The functions *Antecedent*, *UGraph* and *Vectorise* transform bottom-graphs into a form suitable for GNNs by straightforward composition:

Definition 14 (*Graph Transformation*) We define a transformation over $\mathcal{G}$ as follows: $TransformGraph(G) = Vectorise(UGraph(Antecedent((G, \psi))))$.

We now have all the pieces for obtaining graphs suitable for GNNs:

Definition 15 (*BotGNN Graphs*) Given a data instance $e \in \mathcal{E}$ and B, M, d as before, we define $BotGNNGraph_{B,\mathrm{M},d}(e)$, or simply $BotGNNGraph(e) = TransformGraph(BotGraph_{B,\mathrm{M},d}(e))$

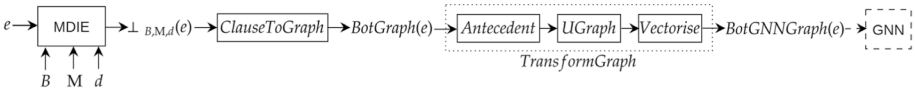
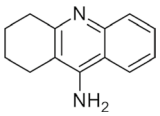


Fig. 3 Construction and use of bottom-graphs for use by GNNs in this paper. We note that constituting the transformation of bottom-graphs are for the GNN implementations used in this paper

Figure 3 summarises the sequence of computations used in this paper. We will use the term *BotGNN* to describe GNNs constructed from BotGNN graphs.

Procedures 1, 2 use the definitions we have introduced to construct and test *BotGNN* models.[13] The procedures assume that data provided as graphs can be represented as definite clauses (Steps 1 in Procedure 1 and 1 in Procedure 2). We illustrate this with an example.

Example 17 The chemical Tacrine is a drug used in the treatment of Alzheimer's disease. It's molecular formula is $C_{13}H_{14}N_2$, and its molecular structure is shown in diagrammatic form below:



One representation of this molecular graph as a definite clause is:

$$
\begin{aligned}
&graph(tacrine) \leftarrow \\
&\quad atom(tacrine, a_1, c), \\
&\quad atom(tacrine, a_2, c), \\
&\quad \vdots \\
&\quad atom(tacrine, a_{13}, c), \\
&\quad atom(tacrine, a_{14}, n), \\
&\quad \vdots \\
&\quad bond(tacrine, a_1, a_2, 1), \\
&\quad bond(tacrine, a_2, a_3, 2), \\
&\quad \vdots
\end{aligned}
$$

More generally, a graph $g = (V, E, \psi, \phi)$ (where V denotes the vertices, E denotes the edges, ψ and ϕ are vertex and edge-label mappings) can be transformed into definite clause of the form $graph(g) \leftarrow Body$, where $Body$ is a conjunction of ground-literals of the form $vertex(g, v_1)$, $vertex(g, v_2)$, ...; $edge(g, e_1)$, $edge(g, e_2)$, ...; $vlabel(g, v_1, \psi(v_1))$, $vlabel(g, v_2, \psi(v_2))$, ...; and $elabel(g, e_1, \psi(e_1))$, $elabel(g, e_2, \psi(e_2))$, ...and so on where $V = \{v_1, v_2, \ldots\}$, $E = \{e_1, e_2, \ldots\}$. More compact representations are possible, but in

[13] In practice, Step 2 of Procedure 1 and Step 2 of Procedure 2 involve some pre-processing that converts the information in BotGNN graphs into a syntactic form suitable for the implementations used. We do not describe these pre-processing details here: they are available as code accompanying the paper.

the experimental section following, we will be using this kind of simple transformation (for molecules: the transformation is done automatically from a standard molecular representation).

Procedure 1: (TrainBotGNN) Construct a $BotGNN$ model, given training data $\{(g_i, y_i)\}_1^N$, where each g_i is a graph and y_i is the class-label for g_i.

Data: Background knowledge B, modes M, depth-limit d, training data $D_{tr} = \{(g_i, y_i)\}_1^N$, and some procedure $TrainGNN$ that trains a graph-based neural network
Result: A $BotGNN$
1. $D'_{tr} = \{\ (g'_i, y_i) : (g_i, y_i) \in D_{tr}$, e_i be a ground definite-clause representing g_i, $g'_i = BotGNNGraph_{B,\mathrm{M},d}(e_i)\ \}$
2. Let $BotGNN = TrainGNN(D'_{tr})$
3. **return** $BotGNN$

Procedure 2: (TestBotGNN) Obtain predictions of a $BotGNN$ model on a data set

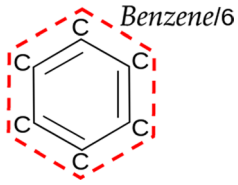
Data: A $BotGNN$ model, background knowledge B, modes M, depth-limit d, and data D consisting of a set of graphs $\{g_i\}_1^N$
Result: $\{(g_i, \hat{y_i})\}_1^N$ where the $\hat{y_i}$ are predictions by $BotGNN$
1. Let $D' = \{(g_i, g'_i) : g_i \in D$, e_i is the definite-clause representation of g_i, $g'_i = BotGNNGraph_{B,\mathrm{M},d}(e_i)\}$
2. Let $Pred = \{(g_i, \hat{y_i}) : (g_i, g'_i) \in D', \hat{y_i} = BotGNN(g')\}$
3. **return** $Pred$

4.4 Note on differences to vertex-enrichment

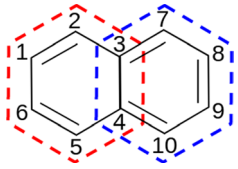
While we defer most related work to a later section (Sect. 6), it is useful to clarify here some differences of BotGNNs with the approach of vertex-enrichment in GNNs (or VEGNNs). These were introduced in Dash et al., (2021c) with the same goal of incorporating symbolic domain-knowledge into GNNs. An immediate difference is in the nature of the graphs handled by the two approaches. Broadly, VEGNNs require data in a graphical form. VEGNNs retain the most of the original graph-structure, but modify the feature-vectors associated with each vertex of the graph (more on this below). BotGNNs on the other hand do not require data to be a graph. Instead, any data representable as a definite clause are reformulated using the bottom-clause into BotGNN graphs. Recall these are bipartite-graphs, in which both vertices and their labels have a different meaning to the graphs in VEGNNs.

A subtler difference between BotGNNs and VEGNNs arises from how the relational information is included within the graphs constructed in each case. The difference is best illustrated by example.

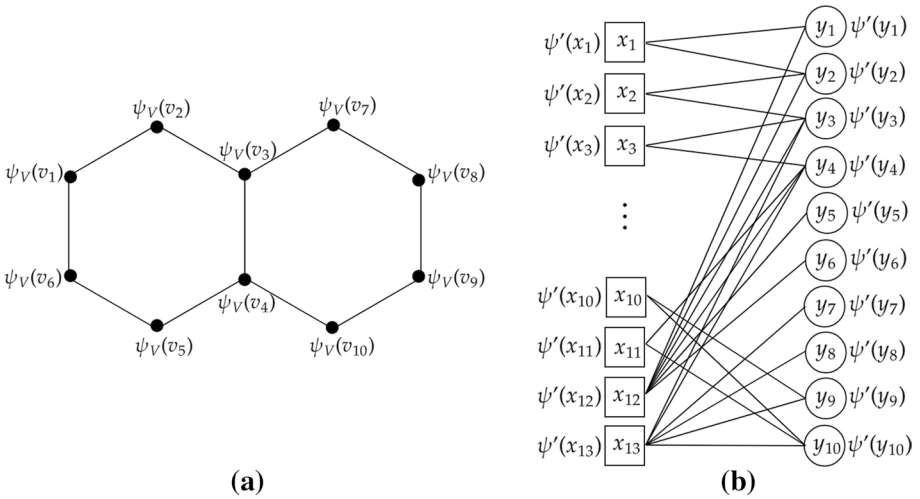
Example 18 Suppose we consider a molecule containing the atoms and bonds shown on the left below, and we want to include the 6-ary relation of a benzene ring (the corresponding hyper-edge is shown dotted on the right below).



In VEGNNs (Dash et al., 2021c), graphs are represented as tuples of the form $(V, E, \sigma, \psi, \phi)$, where V is the set of vertices (here atoms in the molecule); E denotes the edges (bonds in the molecule); σ is a neighbourhood function; ψ denotes an initial vertex-labelling; and ϕ denotes an initial edge-labelling. For each $v \in V$, let $\psi(v)$ be a real-valued vector of finite dimension. In (Dash et al., 2021c), any n-ary relation in domain-knowledge is treated as a hyperedge, where a hyperedge is a set of n vertices in the graph. For any vertex v in a graph, let $h(v)$ denote the set of predicate symbols such that the corresponding hyper-edge contains v. Let $g/1$ be a function that maps sets of predicate-symbols to a fixed-length Boolean-valued vector (a "multi-hot" encoding). Thus, in (Dash et al., 2021c), a VEGNN is a GNN that operates on graphs obtained from labelled graphs of the form $(V, E, \sigma, \psi_V, \phi)$, where $\psi_V(v) = \psi(v) \oplus g(h(v))$ (here $\oplus$ denotes a concatenation operation). In a VEGNN $h(v)$ is $\{Benzene/6\}$ for $v = v_1, \ldots, v_{10}$ in the graph below (representing the compound naphthalene):



Thus, the information that v_3, v_4 are members of 2 different benzene rings is not captured in the VEGNNs vertex-labelleing, and we have to rely on the GNN machinery to re-derive this information from the graph structure (if this information is needed). In a BotGNN on the other hand, the two benzene rings are separate vertices in the bipartite graph, which share edges to vertices representing v_3 and v_4. The broad structure of the VEGNN (only vertex-labels are shown for clarity) and the BotGNN graphs for naphthalene are shown below in (a) and (b) respectively:



$\psi_V/1$ in (a) refers to the vertex-encoding function in (Dash et al.,, 2021c), and ψ' in (b) refers to the function defined in Definition 13. For the experimental data in this paper, the vertex-encoding in Dash et al., (2021c) results in vectors whose dimensions are about 10 times more than the $\psi_V/1$ from Definition 13.

The approach to n-ary relations employed by VEGNNs is thus somewhat akin to a *clique-expansion* of the graph containing vertices for terms. In a clique-expansion, all vertices in a hyper-edge—elements of some n-ary relation—are connected together by a labelled hyper-edge. This can introduce a lot of new edges, and some mechanism is needed to distinguish between multiple occurrences of the same relation (an example is the multiple occurrences of benzene rings above). VEGNNs can be seen as achieving the effect of such a clique-expansion, without explicitly adding the new edges, but they do not address the problem of multiple occurrences. BotGNNs can be seen instead as a *star-expansion* of the graph containing vertices for terms. In such a star-expansion, new nodes denoting the relation are introduced, along with edges between the relation-vertex and the term-vertices that are part of the relation (that is, the hyper-edge). Star-expansions of graphs thus contain 2 kinds of vertices, which is similar to the graph constructed by a BotGNN.

5 Empirical evaluation

5.1 Aims

Our aim in this section is to investigate the following claims:

1. GNNs based on bottom-graphs constructed with domain knowledge (*BotGNN*s) have a higher predictive performance than GNNs that do not use domain knowledge;
2. *BotGNN*s have a higher predictive performance than vertex-enriched GNNs (*VEGNN*s) that employ a simplification to provide domain knowledge to GNNs (Dash et al., 2021c).

Table 1 Dataset summary (The last 3 columns are the average number of X, Y and E in each bottom-graph in a dataset)

# of datasets	Avg # of instances	Avg. of $\|X\|$	Avg. of $\|Y\|$	avg. of $\|E\|$
73	3032	81	42	937

5.2 Materials

5.2.1 Data

For the empirical evaluation of our proposed BotGNNs, we use 73 benchmark datasets arising in the field of drug-discovery. Each dataset represents extensive drug evaluation effort at the NCI[14] to experimentally determine the effectiveness of anti-cancer activity of a compound against a number of cell lines (Marx et al., 2003). The datasets correspond to the concentration parameter GI50, which is the concentration that results in 50% growth inhibition. Each dataset consists of a set of chemical compounds, which are then converted into bottom-graphs.

Each bottom-graph can be represented using $(G, \cdot)$, where $G = (X, Y, E)$, where X represents the vertices corresponding to the relations, Y represents the vertices corresponding to ground terms in the bottom-clause constructed by MDIE, and E represents the edges between X and Y. Table 1 summarises the datasets.

5.2.2 Background knowledge

The initial version of the background knowledge was used in (Van Craenenbroeck et al., 2002; Ando et al., 2006). This BK is a collection of logic programs (written in Prolog) defining almost 100 relations for various functional groups (such as amide, amine, ether, etc.) and various ring structures (such as aromatic, non-aromatic etc.). A functional group is represented by `functional_group/4` predicate and a ring is represented by `ring/4`. There are also higher-level relations defined on the top of the above two relations. These are: the presence of fused rings, connected rings and substructures.

`has_struc(CompoundId, Atoms, Length, Struc)` This relation is *TRUE* if a compound identified by `CompoundId` contains a structure `Struc` of length `Length` containing a set of atoms in `Atoms`.
`fused(CompoundId, Struc1, Struc2)` This relation is *TRUE* if a compound identified by `CompoundId` contains a pair of fused structures `Struc1` and `Struc2` (that is, there is at least 1 pair of common atoms).
`connected(CompoundId, Struc1, Struc2)` This relation is *TRUE* if a compound identified by `CompoundId` contains a pair structures `Struc1` and `Struc2` that are not fused but connected by a bond between an atom in `Struc1` and an atom in `Struc2`.

[14] The National Cancer Institute (https://www.cancer.gov/)

5.2.3 Algorithms and machines

The datasets and the BK are written in Prolog. We use Inductive Logic Programming (ILP) engine, Aleph (Srinivasan, 2001) to construct the bottom-clause using MDIE. A Prolog program then extracts the relations and ground terms from the bottom-clause. We use YAP compiler for execution of all our Prolog programs. These are parsed by UNIX and MATLAB scripts to construct bottom-graph datasets in the format prescribed in (Kersting et al., 2016), which are mainly representations of adjacency matrix, vertex labels (feature vector), class labels, etc.

The GNN variants used here are described in the next section. All the experiments are conducted in a Python environment. The GNN models have been implemented by using the PyTorch Geometric library (Fey and Lenssen, 2019)—a popular geometric deep learning extension for PyTorch (Paszke et al., 2019) enabling easier implementations of various graph convolution and pooling methods.

For all the experiments, we use a machine with Ubuntu (16.04 LTS) operating system, and hardware configuration such as: 64GB of main memory, 16-core Intel Xeon processor, a NVIDIA P4000 graphics processor with 8GB of video memory.

5.3 Method

Let D be a set of data-instances represented as graphs $\{(g_1, y_1), \ldots, (g_N, y_N)\}$, where y_i is a class label associated with the graph g_i. We also assume that we have access to background-knowledge B, a set of modes M, a depth-limit d. Our method for investigating the performance of *BotGNN*s uses is straightforward:

1. Randomly split D into D_{Tr} and D_{Te};
2. Let *BotGNN* be the model from Procedure 1 (TrainBotGNN) with backround knowledge B, modes M, depth-limit d, training data D_{Tr} and some GNN implementation (see below);
3. Let *GNN* be the model from the GNN implementation without background knowledge, and with D_{Tr};
4. Let *VEGNN* be the model using the GNN implementation with vertex-enrichment using the background knowledge B, and with D_{Tr};
5. Let $D'_{Te} = \{g_i : (g_i, y_i) \in D_{Te}\}$
6. Obtain the predictions for D'_{Te} of *BotGNN* using Procedure 2 (TestBotGNN) with background knowledge B, modes M, and depth-limit d;
7. Obtain the predictions for D'_{Te} using *GNN* and *VEGNN*; and
8. Compare the performance of *BotGNN*, *GNN* and *VEGNN*.

The following additional details are relevant. We closely follow the method used in Dash et al., (2021c) for the construction of GNNs. The general workflow involved in GNNs was described in Sect. 2. A diagram of the components involved in implementing that workflow is shown in Fig. 4.

– We have used a 70:30 train-test split for each of the datasets. 10% of the train-set is used as a validation set for hyperparameter tuning.

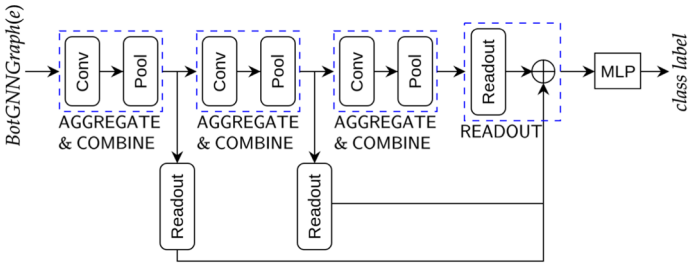


Fig. 4 Components involved in implementing the workflow in Sect. 2 for BotGNN models. 'Conv' and 'Pool' refer to the graph-convolution and graph-pooling operations, respectively. The 'Readout' operation constructs a graph-representation by accumulating information from all the vertex in the graph obtained after the pooling operation. The final graph-representation is obtained in the READOUT block by an element-wise sum (shown as $\oplus$) of the individual graph-representations obtained after each AGGREGATE-COMBINE block. MLP stands for Multilayer Perceptron

- Each GNN consists of three graph convolution blocks and three graph pooling blocks. The convolution and pooling blocks interleave each other (that is, C-P-C-P-C-P) as shown in Fig. 4.
- The convolution blocks can be of one of the following five variants: GCN (Kipf & Welling, 2017), k-GNN (Morris et al., 2019), GAT (Veličković et al., 2018), GraphSAGE (Hamilton et al., 2017), and ARMA (Bianchi et al., 2021). The mathematical details on these graph convolution operations are provided in Appendix B.
- The graph pooling block uses self-attention pooling (Lee et al., 2019) with a pooling ratio of 0.5. We use the graph-convolution formula proposed in Kipf and Welling (2017) for calculating the self-attention scores.
- Due to the large number of experiments (resulting from multiple datasets and multiple GNN variants), the hyperparameters in the convolution blocks are set to the default values within the PyTorch Geometric library.
- We use a hierarchical pooling architecture that uses the readout mechanism proposed by Cangea et al., (2018). The readout block aggregates node features to produce a fixed size intermediate representation for the graph. The final fixed-size representation for the graph is obtained by element-wise addition of the three readout representations.
- The representation length (2 m) is determined by using a validation-based approach. The parameter grid for m is: $\{8, 128\}$, representing a small and a large embedding, respectively.
- The final representation is then fed as input to a 3-layered MLP. We use a dropout layer with a fixed dropout rate of 0.5 after the first layer of MLP.
- The input layer of the MLP contains $2m$ units, followed by two hidden layers with m units and $\lfloor m/2 \rfloor$ units, respectively. The activation function used in the hidden layers is `relu`. The output layer uses `logsoftmax` activation.
- The loss function used is the negative log-likelihood between the target class-labels and the predictions from the model.
- We denote the *BotGNN* variants as: $BotGNN_{1,\ldots,5}$ based on the type of graph convolution method used.
- We use the Adam (Kingma & Ba, 2015) optimiser for training the BotGNNs ($BotGNN_{1,\ldots,5}$). The learning rate is 0.0005, weight decay parameter is 0.0001, the momentum factors are set to the default values of $\beta_{1,2} = (0.9, 0.999)$.

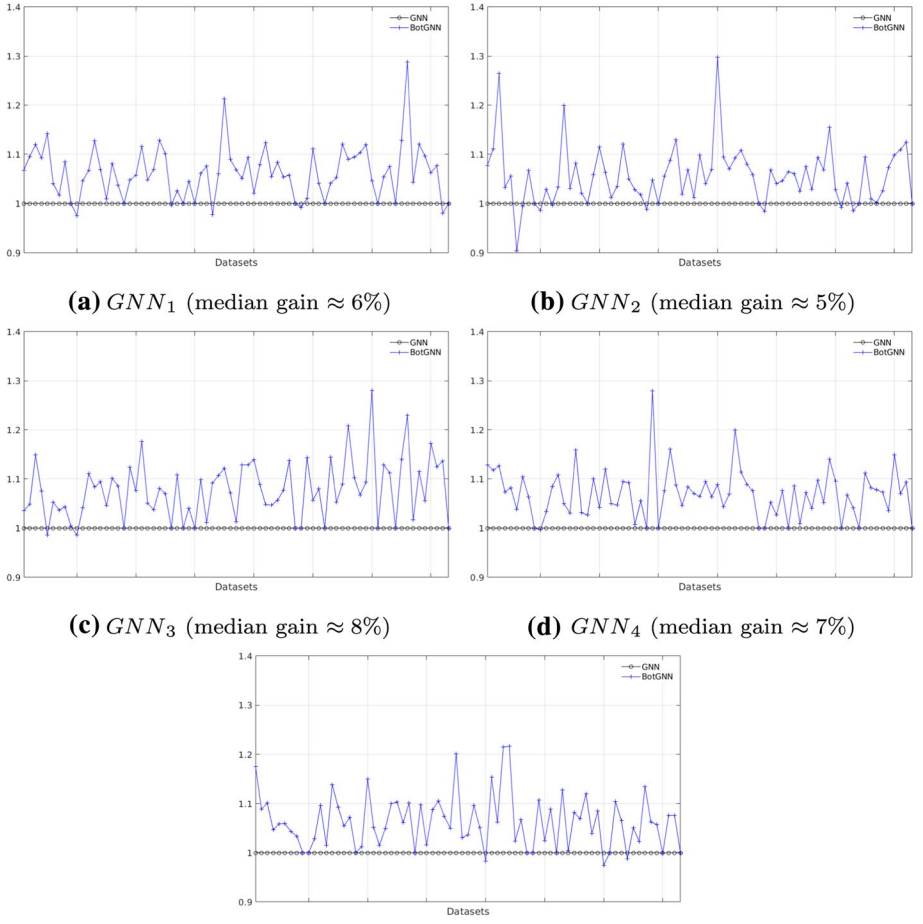
- The maximum number of training epochs is 1000. The batch size is 128.
- We use an early-stopping mechanism (Prechelt, 1998) to avoid overfitting during training. The resulting model is then saved and can be used for evaluation on the independent test-set. The patience period for early-stopping is fixed at 50.
- The predictive performance of a BotGNN model refers to its predictive accuracy on the independent test-set.
- Comparison of the predictive performance of BotGNNs against GNNs and VEGNNs is conducted using the Wilcoxon signed-rank test, using the standard implementation within MATLAB (R2018b).

5.4 Results

The quantitative comparisons of predictive performance of *BotGNN*s against baseline *GNN*s and *VEGNN*s are presented in Table 2. The tabulation shows number of datasets on which *BotGNN* has higher, lower or equal predictive accuracy. The principal conclusions from these tabulations are these: (1) *BotGNN*s perform significantly better than their corresponding counterparts that do not have access to any information other than the atom-and-bond structure of a molecule achieving a gain in predictive accuracy of 5–8% across variants as shown in the qualitative comparison shown in Fig. 5. This is irrespective of the variant of GNN used, suggesting that the technique is able to usefully integrate domain knowledge; (2) *BotGNN*s perform significantly better than *VEGNN*s with access to the same background knowledge. This suggests that *BotGNN*s do more than the vertex-enrichment approach used by *VEGNN*s.

In a previous section (Sect. 4.4) we have described differences between BotGNNs and VEGNNs arising from an encoding of the data into a bipartite graph representation. Possible reasons for this difference in performance are twofold: (1) The GNN variants are unable to use edge-label information. In the VEGNN-style graphs for the data, this information corresponds to the type of bonds. However, this information is contained in vertices associated with the bond-literals in BotGNN-style graphs, which can be used by the GNN-variants; and (2) The potential loss in relational information in VEGNN-style graphs as described in Sect. 4.4. A further difference, not apparent from tabulations of performance is the differences in the feature-vectors associated with each vertex. For the data here, vertex-labels for VEGNNs described in (Dash et al., 2021c) results in each vertex being associated with a 1400-dimensional vector. For BotGNNs, this is about 130.

The significant improvement in predictive performance with the inclusion of symbolic domain-knowledge into GNNs is consistent with similar observations we obtain with multilayer perceptrons (MLPs). For the latter, one well-established way of inclusion of domain-knowledge is through the use of the technique of *propositionalisation* (Lavrač et al., 1991). This represents relational data, like graphs, in the form of some numeric vector (usually, Boolean). For instance, a simple and effective method known as "bottom-clause propositionalisation" or BCP (França et al., 2014) is a propositionalisation approach serving as an extension to one of the pioneering works on neural-symbolic learning systems proposed in (Garcez & Zaverucha, 1999). For a set of (relational) data-instances, BCP obtains a set of (unique) literals from the most-specific (or bottom) clauses constructed by an ILP engine. It then propositionalises each bottom-clause based on the literal set. This process results in each relational data-instance being represented as a Boolean feature-vector. These feature-vectors are then input to an MLP for further processing, allowing the symbolic domain-information available in the bottom-clauses to be easily incorporated into the MLP. Some other related studies have



(a) GNN_1 (median gain $\approx 6\%$)

(b) GNN_2 (median gain $\approx 5\%$)

(c) GNN_3 (median gain $\approx 8\%$)

(d) GNN_4 (median gain $\approx 7\%$)

(e) GNN_5 (median gain $\approx 6\%$)

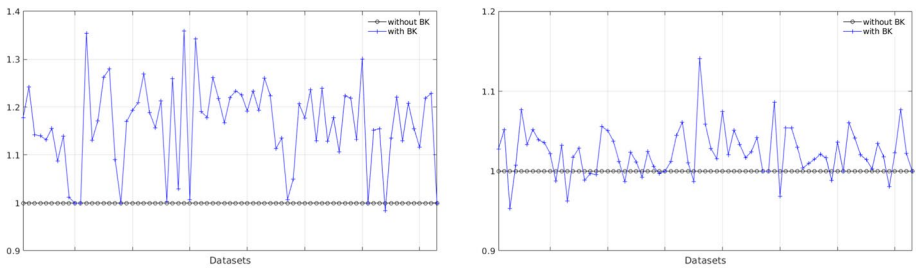
Fig. 5 Qualitative comparison of predictive performance of BotGNNs against Baseline (that is, GNN variants without access to domain-relations). Performance refers to estimates of predictive accuracy (obtained on a holdout set), and all performances are normalised against that of baseline performance (taken as 1). No significance should be attached to the line joining the data points: this is only for visual clarity

shown that the most-specific clauses can also be treated as a source of relational features, described by conjunctions of literals. These relational features can be used to construct a Boolean-vector representation of the (relational) data-instances. These features can form the input feature-vectors for standard statistical models (as in (Saha et al., (2012)) or for multi-layer perceptrons (MLPs). When used with MLPs, the resulting model is called as "deep relational machines" or DRMs, as introduced by Lodhi (2013) and studied extensively in (Dash et al., 2018) and (Dash et al., 2019). In Fig. 6 we also observe gains in performance of MLPs by incorporating domain-knowledge through the use of some form of propositionalisation.

Table 2 Comparison of predictive performance of *BotGNN*s

GNN	Accuracy (*BotGNN*)	
Variant	Higher/lower/equal (p-value)	
	GNN	*VEGNN*
1	59/5/9 (< 0.001)	54/11/8 (< 0.001)
2	59/8/6 (< 0.001)	61/9/3 (< 0.001)
3	61/2/10 (< 0.001)	54/10/9 (< 0.001)
4	63/1/9 (< 0.001)	55/11/7 (< 0.001)
5	60/4/9 (< 0.001)	52/9/12 (< 0.001)

The tabulations are the number of datasets on which *BotGNN* has higher, lower or equal predictive accuracy (obtained on a holdout set) than *GNN* and *VEGNN*. Statistical significance is computed by the Wilcoxon signed-rank test



(a) DRM **(b)** BCP+MLP

Fig. 6 Improvements in predictive performance of MLPs, when provided with domain-knowledge through propositionalisation. Baselines ("1") are the models without domain-knowledge. The DRM here is an MLP that uses simple Boolean propositions indicating the presence or absence of relations provided as domain-knowledge. The structure and parameters of the MLP are obtained using the Adam optimiser (Kingma & Ba, 2015). BCP+MLP use Boolean propositions constructed using the bottom-clause propositionalisation method in França et al., (2014). "MLP" refers to the use of these features by a multi-layer perceptron. The structure and parameters for the MLP are obtained using the same approach as the DRM. The domain-knowledge is the same as that used for the construction of BotGNNs

5.5 Some additional results

We turn now to two questions that are not within the scope of the experimental goals listed in Sect. 5.1, but are nevertheless of practical interest (details relevant to the results are in Appendix C). First, the question of whether incorporating domain-knowledge using the route of bottom-graphs and GNNs is better than propositions and MLPs? A straightforward comparison of the *BotGNN* models against those used in Fig. 6 would seem to suggest that the answer is "yes" (Table 3). However, we caution against drawing such a conclusion for at least the following reasons: (a) Inclusion of propositions based on stochastic sampling of complex relational features in (Dash et al., 2019) can result in significantly better DRM models. It is possible also that BCP could be augmented with the same sampling methods to yield more informative propositions; (b) It is also possible that a *BotGNN* obtained with access to sampled relational features could improve its performance over what is shown here. We note that propositionalisation is not confined to the use of MLPs: so it would be

Table 3 Comparison of predictive performance of BotGNN with DRM and BCP+MLP

GNN	Accuracy (*BotGNN*)	
Variant	Higher/lower/equal (p-value)	
	DRM	BCP+MLP
1	47/22/4 (< 0.001)	58/10/5 (< 0.001)
2	41/29/3 (0.065)	58/11/4 (< 0.001)
3	45/19/9 (< 0.001)	61/6/6 (< 0.001)
4	50/19/4 (< 0.001)	62/6/5 (< 0.001)
5	51/16/6 (< 0.001)	60/6/7 (< 0.001)

The tabulations are the number of datasets on which *BotGNN* has higher, lower or equal predictive accuracy (obtained on a holdout set) than DRM and BCP+MLP. DRM and BCP+MLP refer to the models in Fig. 6

Table 4 Characterisation of vector-representation used for model-construction by BotGNNs, DRMs and BCP+MLP. Minimum/maximum values of the range are only shown to 3 meaningful digits (the actual values are not relevant here)

Method	Vector representation	Vector dimension (range)
BotGNN	Real, dense	16–256
DRM	Boolean, sparse	1400–1400
BCP+MLP	Boolean, very sparse	18,000–52,000

The graph-representations (also, called graph-embeddings) for BotGNNs are constructed internally by the GNN. By "sparse" we mean that there are many 0-values, and by "very sparse", we mean the values are mostly 0

not be surprising if the BotGNN performance was bettered by some ML method using the data provided to the DRM or BCP+MLP.

A more useful difference between the BotGNN approach and propositionalisation is that techniques relying on the latter usually separate the feature- and model-construction steps. A *BotGNN*, like any GNN, constructs a vector-space embedding for the graphs it is provided. However, this embedding is obtained as part of an end-to-end model construction process. This can be substantially more compact than the representation used by methods that employ a separate propositionalisation step (see Table 4).

The second question of practical interest is how a BotGNN's performance compares to an ILP learner that uses bottom-clauses directly. Table 5(a) shows a routine comparison against the Aleph system (Srinivasan, 2001) configured to perform greedy set-covering for identifying rules using bottom-clauses (in effect, a Progol-like approach: see (Muggleton, 1995)). Again, we caution against drawing the obvious conclusion, since the results are obtained without attempting to optimise any parameters of the ILP learner (only the minimum accuracy of clauses was changed from the default setting of 1.0 to 0.7: this latter value has been shown to be more appropriate in many previous experimental studies with Aleph). A better indication is in Table 5(b), which compares BotGNN performance on older benchmarks for

Table 5 Comparison of predictive performance of BotGNNs with an ILP learner (Aleph system): (a) Without hyperparameter tuning in Aleph; (b) With hyperparameter tuning

GNN	Accuracy (*BotGNN*)	
Variant	Higher/lower/equal (p-value)	
(a)		
1	62/7/4 (< 0.001)	
2	60/9/4 (< 0.001)	
3	61/7/5 (< 0.001)	
4	62/6/5 (< 0.001)	
5	62/4/7 (< 0.001)	
Dataset	ILP	BotGNN
(b)		
DssTox	0.73	0.76
Mutag	0.88	0.89
Canc	0.58	0.64
Amine	0.80	0.84
Choline	0.77	0.72
Scop	0.67	0.65
Toxic	0.87	0.85

In (a), the tabulations are the number of datasets on which *BotGNN* has higher, lower or equal predictive accuracy (obtained on a holdout set) than the ILP learner. In (b), each entry is the average of the accuracy obtained across 10-fold validation splits (as in (Srinivasan et al., 2003))

which ILP results after parameter optimisation are available. These suggest that BotGNN performance to be comparable to an ILP approach with optimised parameter settings.[15]

6 Related work

One of the oldest approaches that incorporate domain knowledge into feature-based machine learning is LINUS (Lavrač et al., 1991), which proposed the idea of an attribute-value learning system based on a method called 'propositionalisation'. This is a simple and effective approach to construct a numeric feature-vector representation for domain-relations (Lavrač et al., 2021), which can then be used as input features for deep neural networks. One such example is Deep Relational Machines (DRMs: Lodhi, 2013). Some large-scale studies show that propositionalisation of relational (first-order) features results in an effective way of incorporating symbolic domain-knowledge into deep networks (Dash et al., 2018, 2019). The pioneering work on connectionist inductive learning and logic programming (CIL2P) by Garcez and Zaverucha (1999) is extended in an interesting idea of propositionalisation called 'Bottom Clause Proposionalisation (BCP)' (França et al., 2014). BCP converts a bottom-clauses in ILP into Boolean vectors, which can then be used to learn a neural network more efficiently and faster than its predecessor. Although propositionalisation approaches are a simple and straight-forward way of constructing a feature-vector that encodes both relational data and domain-knowledge, the present forefront of deep networks dealing directly with relational data (graphs) are GNNs, where we can represent relational knowledge in neural networks directly without a propositionalisation step.

Other approaches of incorporating domain-knowledge include a modification to the loss function that is optimised during training a deep network (Xu et al., 2018; Fischer et al., 2019). In these approaches, the primary structure of the underlying deep network stays roughly unaltered. One such example is: Domain-adapted neural network (DANN) that introduces an (additional) domain-based loss term to the neural network loss function. In particular, the domain-specific rules are approximated using two kinds of constraints: approximation constraint and monotonicity constraint. It is claimed that incorporating domain-knowledge in this manner enforces the network not only to learn from the available training data but also domain-specific rules (Muralidhar et al., 2018).

Under the category of statistical relational learning (SRL), there are some interesting proposals to learn from relational data (often, graph-structured), and symbolic domain-knowledge. For instance, the work on kLog by Frasconi et al. (2014) introduced a method called 'graphicalisation' to convert relational structures (first-order logic interpretations) to graphs. By this conversion, it enables the use of graph-kernel methods, which measures the similarity between two graphs.[16] kLog generates propositional features (based on the graph kernel) for use in SRL. kLog and BotGNN share some common characteristics: (1)

[15] We note that parameter screening and optimisation is not routinely done in ILP. In (Srinivasan and Ramakrishnan, 2011) it is noted: "Reports in the [ILP] literature rarely contain any discussion of sensitive parameters of the system or their values. Of 100 experimental studies reported in papers presented between 1998 and 2008 to the principal conference in the area, none attempt any form of screening for relevant parameters."

[16] Graph Kernel: A kernel function compares substructures of graphs that are computable in polynomial time (Vishwanathan et al., 2010).

at the first level of modelling, they both construct bipartite graphs representing the relational instances: kLog constructs undirected bipartite graphs that are related to 'Probabilistic Entity-Relationship model' (Heckerman et al., 2007), whereas our approach here constructs directed bipartite graph, and the construction is different from the former representation to a great extent, relying heavily on a vertex-labelling based on the concept of a mode-language used within ILP; (2) at the second level of learning, kLog employs graph kernels to construct propositional features and uses those to learn a statistical learner, whereas BotGNN leverages GNNs, which are superior to graph kernels in practice (Du et al., 2019). Though there are some similarities and differences between kLog and BotGNN, one should note that the primary aim of the former is a proposal for a language for statistical relational learning, whereas our primary aim is to propose a principled way to incorporate symbolic domain-knowledge into GNNs.

Falling under the same umbrella of SRL is work on the integration of relational learning with GNNs (Šourek et al., 2021), which demonstrates how simple relational logic programs can capture advanced graph convolution operations in a tightly integrated manner, requiring the use of a language of Lifted Relational Neural Networks (LRNNs) (Sourek et al., 2018). The integration of logic programs and GNNs in this manner results in an interesting GNN-based neuro-symbolic model (Lamb et al., 2020). The input representation for LRNN is a weighted logic program or *template*, which is mainly different from the input representation used for BotGNNs.

Knowledge-graphs (KGs) are a rich source of domain-knowledge and are a representation of binary relations. In a KG, each vertex represents an entity. The relation between two vertices is represented as an edge between them. In the last few years, several methods have attempted to incorporate the information (relations) encoded in KGs into deep neural networks. For instance, Schlichtkrull et al. (2018) introduced Relational Graph Convolutional Networks (R-GCNs) that models the information exchange among different entities via the relations in a KG with the help of the message-passing in a GCN. The approach was intended for entity classification and link prediction (discovering the missing relation between two entities). R-GCNs are related to our BotGNNs in one sense that they are able to model multi-relational information. However, R-GCNs are restricted to KGs to deal only with binary relations, albeit they could be extended to go beyond binary relations using hypergraph neural networks (Feng et al., 2019; Bai et al., 2021). A method proposed for incorporating knowledge-graphs into deep networks is termed as "knowledge-infused learning" (Sheth et al., 2019; Kursuncu et al., 2020). The work examines techniques for incorporating relations at various layers of deep networks (the authors categorise these as: shallow, semi-deep and deep infusion). A GNN can directly operate on knowledge-graphs for constructing node and graph representations that are useful for further learning (Wang et al., 2019). We note that BotGNNs could be seen as performing a generalised form of knowledge-infused learning, in which: (a) Data can contain n-ary relations; and (b) Domain-knowledge can encode arbitrary relations of possible relevance to the data. In the original work on knowledge-infusion, data are knowledge-graphs that only employ binary relations, and there is no possibility of including additional domain-knowledge (that is, $B = \emptyset$).

A method (Xie et al., 2019) proposed that the symbolic knowledge can be represented as formulas in Conjunctive Normal Form (CNF) and decision-Deterministic Decomposable Negation Normal Form (d-DNNF). These formulas can naturally be viewed as graph

structures. Learning on these graph structures is then carried out using a GNN. A recently proposed method uses the idea of treating symbolic domain-relations as hyperedges (Dash et al., 2021c). These hyperedges can then be used to construct the labelling for the nodes of a graph using a method called 'vertex-enrichment', which is a simplified approach to incorporate symbolic domain-knowledge into GNNs. This form of GNNs are called Vertex-Enriched GNNs (VEGNNs). A detailed note on the differences between VEGNNs and BotGNNs are already provided in an earlier section. Briefly, there are three main differences: (1) VEGNNs require data to be represented as graphs; whereas, BotGNNs can deal with any data that can be represented as definite clauses; (2) VEGNNs introduce symbolic domain-relations by modifying only the vertex-labelling of the graphs while maintaining the original graph structure of the data; whereas, BotGNNs combine data and background knowledge (using MDIE) to construct BotGNN graph representations, which are bipartite graphs; and (3) VEGNNs do not allow some of the crucial information about a vertex to be automatically conveyed via the vertex-labellings, for example, a vertex is a member of two different benzene rings; whereas, in BotGNNs this information is readily available from the bipartite graph-structure. A recent systematic review on various methods of incorporating domain-knowledge into deep neural networks categorises domain-knowledge into two classes of constraints: logical and numerical (Dash et al., 2021b). Our present work falls under the former category and aims to constrain the structure of the graph (here, bipartite graph). An extended version of this survey can be found in (Dash et al., 2021a) which categories the methods of incorporating domain-knowledge into deep networks based on whether (a) the input-representation is changed, (b) the loss-function is changed, or (c) the structure of the deep network is changed. Our proposed BotGNN approach falls under the first category where each (relational) data-instance is changed to a bipartite graphs representation.

7 Conclusions

The Domain-Knowledge Grand Challenge in (Stevens et al., 2020) calls for systematically incorporating diverse forms of domain knowledge. In this paper, we have proposed a systematic way of incorporating domain-knowledge encoded in a powerful subset of first-order logic. The technique explicitly addresses the requirement in (Stevens et al., 2020) of "extending" the raw data for a class of neural networks that deal with graphs. The significant improvements in performance that we have observed support the statement on the importance of the role of domain knowledge. We have also provided additional results that suggest that the technique may be doing more than a simple "propositionalisation".

On the face of it, it would seem that logic programs are necessary to construct the BotGNNs proposed here. We distinguish here between the principle we have proposed and its implementation using logic programs. The construction of a most-specific clause, as is done by MDIE, is necessary for the construction of a BotGNN. However, the construction of this clause need not be done by using logic programming technology: it has been shown, for example, in (Bravo et al., 2005) how this same formula can be obtained entirely using operations on relational databases. In many ways, such an implementation would be more convenient, since the information in modes can be incorporated (and even augmented) by the schema of a relational database. Looking further ahead, it is possible that domain-knowledge could even be communicated as unstructured statements in a natural language. However, for problems of scientific discovery it would appear to be more useful if domain-knowledge was represented in some structured, mathematical form.

We do not view BotGNNs as an alternative to ILP. Instead, the purpose in this paper is to show that techniques developed in ILP can be used to incorporate symbolic domain knowledge into deep neural networks, with significant improvement in predictive performance. In fact, relational definitions found by an ILP engine may be able to improve a BotGNN's performance further, possibly by augmenting bottom-graphs. This can be done by either simple inclusion of the new relations as background knowledge, or through an extension of clause-graphs from bipartite to more general k-partite graphs.

The linking of symbolic and neural techniques allows the possibility of providing logical explanations for predictions made by the neural model. This potential has long been recognised, and demonstrated (see for example: (Besold et al., 2017; Garcez et al., 2019)). Here, we expect the relationships shown in Remark 4 (see Appendix A) may open the interesting possibility of linking clausal explanations to GNNs. The improvement in predictive performance of GNNs by the incorporation of domain knowledge is a necessary part of their use as tools for scientific discovery. But that is not sufficient: explanations in terms of relevant domain-concepts will also be needed. We intend to look at this in future.

Appendix A: Some properties of clause-graphs

We note the following properties about clause-graphs. For these properties, we assume background knowledge B, a set of modes M, and a depth-limit d as before. Clause-graphs are elements of the set $\mathcal{G}$ and are structures of the form $((X, Y, E), \psi)$ where (X, Y, E) are bipartite graphs from the set $\mathcal{B}$ (see Sect. 4.1). We assume $\mathcal{G}$ contains the element $CG_\top = ((\emptyset, \emptyset, \emptyset), \emptyset)$. We also define the following equality relation over elements of $\mathcal{G}$: $(X_i, Y_i, E_i), \psi_i) = (X_j, Y_j, E_j), \psi_j)$ iff $X_i = X_j$, $Y_i = Y_j$, $E_i = E_j$ and $\psi_i = \psi_j$. Also, given a clause $C = \{l_1, \ldots, l_m, \neg l_{m+1}, \ldots, \neg l_k\}, 1 \leq m < k$, Λ_C is the set $\{l_1, \ldots, l_{m+1}, \ldots, l_k\}$.

Definition 16 ($\preceq_{cg}$) Let $CG_1 = (G_1, \psi_1)$, $CG_2 = (G_2, \psi_2)$ be elements of $\mathcal{G}$, where $G_1 = (X_1, Y_1, E_1)$ and $G_2 = (X_2, Y_2, E_2)$ Then $CG_1 \preceq_{cg} CG_2$ iff: (a) $X_1 \subseteq X_2$; (b) $Y_1 \subseteq Y_2$; (c) $E_1 \subseteq E_2$; and (d) $\psi_1 \subseteq \psi_2$.

Proposition 1 *$\langle \mathcal{G}, \preceq_{cg} \rangle$ is partially ordered.*

Proof In the following, let $CG = ((X, Y, E), \psi)$, and $CG_i = ((X_i, Y_i, E_i), \psi_i)$.

Reflexive: If $CG \in \mathcal{G}$ then $CG \preceq_{cg} CG$. This follows trivially since $X \subseteq X, Y \subseteq Y, E \subseteq E$ and $\psi \subseteq \psi$.
Anti-Symmetric: Let $CG_1, CG_2 \in \mathcal{G}$. If $CG_1 \preceq_{cg} CG_2$ and $CG_2 \preceq_{cg} CG_1$ then $CG_1 = CG_2$. Since $CG_1 \preceq_{cg} CG_2$, and $CG_2 \preceq_{cg} CG_1$ $X_1 \subseteq X_2$ and $X_2 \subseteq X_1$. Therefore $X_1 = X_2$. Similarly $Y_1 = Y_2$, $E_1 = E_2$ and $\psi_1 = \psi_2$, and therefore $CG_1 = CG_2$;
Transitive: Let $CG_1, CG_2, CG_3 \in \mathcal{G}$. If $CG_1 \preceq_{cg} CG_2$ and $CG_2 \preceq_{cg} CG_3$ then $CG_1 \preceq_{cg} CG_3$. Since $CG_1 \preceq_{cg} CG_2$ and $CG_2 \preceq_{cg} CG_3$ then $X_1 \subseteq X_2$ and $X_2 \subseteq X_3$. Therefore $X_1 \subseteq X_3$. Similarly, $Y_1 \subseteq Y_3$, $E_1 \subseteq E_3$ and $\psi_1 \subseteq \psi_3$. Therefore, $CG_1 \preceq_{cg} CG_3$. □

For $CG_1, CG_2 \in \mathcal{G}$, if $CG_1 \preceq_{cg} CG_2$, then we will say CG_1 is more general than CG_2. We note without formal proof that if $CG \in \mathcal{G}$ then $CG_\top \preceq_{cg} CG$.

Remark 2 We note the following consequences of the Definitions 7–9, and Definition 16 above:

(i) Let $C, D \in \mathcal{L}_{B,\mathrm{M},d}$, $CG_1 = ClauseToGraph(C)$, and $CG_2 = ClauseToGraph(D)$ where: $CG_1 = ((X_1, Y_1, E_1), \psi_1)$ and $CG_2 = ((X_2, Y_2, E_2), \psi_2)$. If $(X_1 \subseteq X_2)$ then $CG_1 \preceq_{cg} CG_2$. By construction, $X_1 \subseteq X_2$ iff $Lits(C) \subseteq Lits(D)$. It follows that $Terms(C) \subseteq Terms(D)$, and $Y_1 \subseteq Y_2$. Since E_1 contains all the relevant arcs between X_1 and Y_1 and E_2 contains all the relevant arcs between X_2 and Y_2, E_2 will contain all the elements of E_1. Since h_x, h_y are bijections, $\psi_1 \subseteq \psi_2$. That is $CG_1 \preceq_{cg} CG_2$;

(ii) Let $C, D \in \mathcal{L}_{B,\mathrm{M},d}$, $CG_1 = ClauseToGraph(C) = ((X_1, Y_1, E_1), \psi_1)$ and $CG_2 = ClauseToGraph(D) = ((X_2, Y_2, E_2), \psi_2)$. Let $\mathcal{LM}_1$ be the set of $\lambda\mu$-sequences for C and $\mathcal{LM}_2$ be the set of $\lambda\mu$-sequences for D. If $\mathcal{LM}_1 \subseteq \mathcal{LM}_2$ then $CG_1 \preceq_{cg} CG_2$. It is evident that $Lits(C) \subseteq Lits(D)$. Therefore $X_1 \subseteq X_2$, and from the observation (i) above, $CG_1 \preceq_{cg} CG_2$).

Lemma 1 (Lits) *The function* $Lits : \mathcal{L}_{B,M,d} \rightarrow 2^{LM}$ *(defined in Definition 7) is well-defined. That is, if* $C = D$, *then* $Lits(C) = Lits(D)$.

Proof Assume the contrary. That is, $C = D$ and $Lits(C) \neq Lits(D)$. Since $C = D$, $\Lambda_C = \Lambda_D$. Further, since $Lits(C) \neq Lits(D)$, for some $\lambda_i \in \Lambda_C, \Lambda_D$, there must exist $\mu_i \in \mathrm{M}$ s.t. $(\lambda_i, \mu_i) \in Lits(C)$ and $(\lambda_i, \mu_i) \notin Lits(D)$ or *vice versa*. This is not possible since $Lits(C)$ and $Lits(D)$ contain all $\lambda\mu$-sequences for C, D. □

Lemma 2 *Let* $C, D \in \mathcal{L}_{B,\mathrm{M},d}$. *Let* $CG_1 = ClauseToGraph(C)$ *and* $CG_2 = ClauseToGraph(D)$. *if* $C = D$ *then* $CG_1 = CG_2$.

Proof The result holds trivially if $C = D = \emptyset$; and we consider $C, D \neq \emptyset$. Let $CG_1 = ((X_1, Y_1, E_1), \psi_1)$ and $CG_2 = ((X_2, Y_2, E_2), \psi_2)$. Since $C = D$, by Lemma 1 $Lits(C) = Lits(D)$. From Definition 8, $(Terms(C) = Terms(D))$ iff $(Lits(C) = Lits(D))$. From Definition 9, $(X_1 = X_2)$ iff $(Lits(C) = Lits(D))$ and $(Y_1 = Y_2)$ iff $(Terms(C) = Terms(D))$. If $(X_1 = X_2)$ and $(Y_1 = Y_2)$ then $(E_1 = E_2)$. Since h_x, h_y are bijections, $\psi_1 = \psi_2$. That is, $CG_1 = CG_2$. □

Proposition 2 (ClauseToGraph) *The function* $ClauseToGraph : \mathcal{L}_{B,\mathrm{M},d} \rightarrow \mathcal{G}$ *(defined in Definition 9) is injective*.

Proof Let C and D in $\mathcal{L}_{B,\mathrm{M},d}$, and $CG_1 = ClauseToGraph(C)$ and $CG_2 = ClauseToGraph(D)$. We need to show that if $CG_1 = CG_2$ then $C = D$.

Let $CG_1 = (G_1, \psi_1)$ and and $CG_2 = (G_2, \psi_2)$, where $G_1 = (X_1, Y_1, E_1)$ and $G_2 = (X_2, Y_2, E_2)$. Since $CG_1 = CG_2$, $(G_1, \psi_1) = (G_2, \psi_2)$. That is, $X_1 = X_2$, $Y_1 = Y_2$ and $\psi_1 = \psi_2$. Suppose $C \neq D$. Then, either there is some literal in C that is not in D or *vice versa*. Let $\lambda_i \in C$, and $\lambda_i \notin D$. Let λ_i be the corresponding literal in Λ_C, and $\lambda_i \notin \Lambda_D$. Then since $C \in \mathcal{L}_{B,\mathrm{M},d}$ there must be at least one $\mu_i \in \mathrm{M}$ s.t. $(\lambda_i, \mu_i) \in Lits(C)$. Let $x = h_x((\lambda_i, \mu_i)) \in X_1$. Since h_x is a bijection, and $\lambda_i \notin D$, there will be no other λ and μ such that $h_x((\lambda, \mu)) = x$. Hence $x \notin X_2$. This is a contradiction, since $X_1 = X_2$. Similarly for $\lambda_i \in D$ and $\lambda_i \notin C$.

□

Proposition 3 (Left-Inverse) *ClauseToGraph*$(\cdot)$ *has a left-inverse*.

Proof We show that there is a function $GraphtoClause : \mathcal{G} \rightarrow \mathcal{L}_{B,\mathrm{M},d}$ s.t. for all $C \in \mathcal{L}_{B,\mathrm{M},d}$, $GraphToClause(ClauseToGraph(C)) = C$.

Let $CG = ClauseToGraph(C)$. So, $CG = (G, \psi)$, where $G = (X, Y, E)$. For each $x_i \in X$:

1. Let $L^+ = \{\lambda_i : x_i \in X, \psi(x_i) = (\lambda_i, \mu_i), \mu_i = modeh(\cdot)\}$;
2. Let $L^- = \{\neg\lambda_i : x_i \in X, \psi(x_i) = (\lambda_i, \mu_i), \mu_i = modeb(\cdot)\}$

Let $GraphToClause(CG) = C'$ where $C' = L^+ \cup L^-$. We claim $C = C'$. Assume $C \neq C'$. Then there must be some literal $l_i \in C$ s.t. $l_i \notin C'$ (or *vice versa*). Let the corresponding literal in Λ_C be λ_i. Since $C \in \mathcal{L}_{B,\mathrm{M},d}$, there must be some $\lambda\mu$-sequence (Definition 5) for C s.t. some $(\lambda_i, \mu_i) \in Lits(C)$ (Definition 7) and $h_x((\lambda_i, \mu_i)) \in X$ (Definition 9). Then, by the construction above, $l_i \in C'$, which is a contradiction. Suppose $l_i \in C'$ and $l_i \notin C$. Then there cannot be any (λ_i, μ_i) s.t. $h_x((\lambda_i, \mu_i) \in X$. By construction, $l_i \notin C'$, which is a contradiction. Therefore there is no $l_i \in C$ and $l_i \notin C'$, or *vice versa*, and $C = C'$.

Remark 3 We note the following without formal proofs:

(i) In Definition 10, if there exists a unique $\bot_{B,\mathrm{M},d}(e) \in \mathcal{L}_{B,\mathrm{M},d}$ then $BotGraph_{B,\mathrm{M},d}(e)$ is unique. The proof follows from $BotGraph_{B,\mathrm{M},d}(e) = ClauseToGraph(\bot_{B,\mathrm{M},d}(e))$.

(ii) In Definition 11, $Antecedent : \mathcal{G} \rightarrow \mathcal{G}$ is well-defined. That is, if $Antecedent(CG_1) \neq Antecedent(CG_2)$ then $CG_1 \neq CG_2$. Again the proof follows from the contrapositive which is easily seen to hold. Also, we note that *Antecedent* is many-to-one, that is is possible that $CG_1 \neq CG_2$, and $Antecedent(CG_1) = Antecedent(CG_2)$.

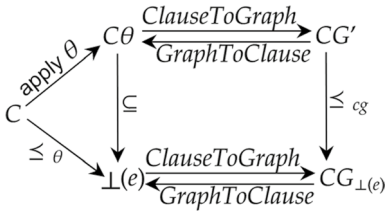
(iii) In Definition 12, $UGraph : \mathcal{G} \rightarrow \mathcal{G}$ is well-defined. That is, if $UGraph(CG_1) \neq UGraph(CG_2)$ then $CG_1 \neq CG_2$. This follows from the contrapositive which is easily shown to hold (that is, if $CG_1 = CG_2$ then $UGraph(CG_1) = UGraph(CG_2)$).

Finally, we relate the clausal explanations found by some MDIE systems using the ordering $\preceq_\theta$ defined over clauses in (Plotkin, 1970).[17] Given background knowledge B and a clause e, we will say a clause C is a clausal explanation for e if $B \cup \{C\} \models e$.

Remark 4 (Relation to Clausal Explanations) Let $\bot_{B,\mathrm{M},d}(e)$ be the ground most-specific definite-clause using MDIE. Let C be a clause (not necessarily ground). We show the following: If $C\theta \subseteq \bot_{B,\mathrm{M},d}(e)$ and $C\theta \in \mathcal{L}_{B,\mathrm{M},d}$, then there exists a clause-graph CG' s.t. $CG' \preceq_{cg} ClauseToGraph(\bot_{B,\mathrm{M},d}(e)$ and $GraphToClause(CG') = C\theta$.

Denoting $\bot_{B,\mathrm{M},d}(e)$ as $\bot(e)$ and $ClauseToGraph(\bot_{B,\mathrm{M},d})$ as $CG_{\bot(e)}$ the relationships described in this remark is shown diagrammatically as:

[17] $C_1 \preceq_\theta C_2$ if there exists some substitution θ s.t. $C_1\theta \subseteq C_2$. By convention C_1 is said to be more-general than C_2, and C_2 is said to be more-specific than C_1. It is known that if $C_1 \preceq_\theta C_2$, then $C_1 \models C_2$.



Let $ClauseToGraph(\bot_{B,\mathrm{M},d}(e)) = (G, \psi)$, with $G = (X, Y, E)$. In the following, $Pos(l) = p$ if $l = \neg p$ is a negative literal, otherwise $Pos(l) = l$. Consider the structure $CG' = (G', \psi')$, with $G' = (X', Y', E')$ obtained as follows.

(a) $X' = \{x_i : x_i \in X, l_i \in C\theta, \lambda_i = Pos(l_i), \psi(x_i) = (\lambda_i, \mu_i)\}$;
(b) $E' = \{(x_i, y_j) : x_i \in X', (x_i, y_j) \in E\} \cup \{(y_j, x_i) : x_i \in X', (y_j, x_i) \in E\}$;
(c) $Y' = \{y_j : (x_i, y_j) \in E' \text{ or } (y_j, x_i) \in E'\}$;
(d) For $v \in X' \cup Y'$, $\psi'(v) = \psi(v)$

It is evident that G' is a directed bipartite graph, and ψ' is defined for every vertex in G'. So, $CG' \in \mathcal{G}$. By construction, CG' has the following properties: (i) $X' \subseteq X$ and $Y' \subseteq Y$; (ii) $E' \subseteq E$; and (iii) $\psi' \subseteq \psi$. Therefore $CG' \preceq_{cg} (G, \psi)$. Since the vertices in X' are obtained using only the literals in $C\theta$, it follows that $GraphToClause(CG) = C\theta$.

Since $C\theta \subseteq \bot_{B,\mathrm{M},d}(e)$, $C\theta \models \bot_{B,\mathrm{M},d}(e)$. Further, since $C \preceq_\theta C\theta$, $C \models C\theta$. It follows that $B \cup C \models B \cup \bot_{B,\mathrm{M},d}(e)$. Since $B \cup \bot_{B,\mathrm{M},d}(e) \models e$, then $B \cup C \models e$. That is, C is a clausal explanation for e.

Appendix B: Implementation details on GNNs

We provide some basic mathematical details on how various graph convolutions and pooling operations are implemented in the five different GNN variants considered in our work. This section is meant for completeness only. The reader should refer to the primary sources of these implementations for a detailed information on each of these approaches.

B.1 Graph convolution

For each graph convolution method, we only describe how the AGGREGATE-COMBINE procedure (as described in Sect. 2) is implemented.

Variant 1: GCN

Based on the spectral-based graph convolution as proposed by Kipf and Welling (2017), this graph convolution uses a layer-wise (or iteration-wise) propagation rule for a graph with N vertices as:

$$\mathbf{H}^{(k)} = \sigma\left(\tilde{D}^{-\frac{1}{2}}\tilde{A}\tilde{D}^{-\frac{1}{2}}\mathbf{H}^{(k-1)}\boldsymbol{\Theta}^{(k-1)}\right) \tag{1}$$

where, $H^{(k)} \in \mathbb{R}^{N\times D}$ denotes the matrix of vertex representations of length D, $\tilde{A} = A + I$ is the adjacency matrix representing an undirected graph G with added self-connections, $A \in \mathbb{R}^{N\times N}$ is the graph adjacency matrix, I_N is the identity matrix, $\tilde{D}_{ii} = \sum_j \tilde{A}_{ij}$, and $\boldsymbol{\Theta}^{(k-1)}$ is the iteration-specific trainable parameter matrix, $\sigma(\cdot)$ denotes the activation function e.g. $\mathrm{ReLU}(\cdot) = \max(0, \cdot)$, $\mathbf{H}^{(0)} = \mathbf{X}$, $\mathbf{X}$ is the matrix of feature-vectors of the vertices, where each vertex i is associated with a feature-vector X_i.

Variant 2: *k*-GNN

This graph convolution passes messages (vertex feature-vectors) directly between subgraph structures inside a graph (Morris et al., 2019). At iteration k, the feature representation of a vertex is computed by using

$$h_u^{(k)} = \sigma\left(h_u^{(k-1)} \cdot \Theta_1^{(k)} + \sum_{v\in\mathcal{N}(u)} h_v^{(k-1)} \cdot \Theta_2^{(k)}\right) \tag{2}$$

where, h_u^k denotes the vertex-representation of a vertex u at iteration k, $\mathcal{N}$ denotes the neighborhood function, σ is a non-linear transfer function applied component wise to the function argument, Θs are the layer-specific learnable parameters of the network.

Variant 3: GAT

This variant is based on aggregating information from neighbours with attention. This approach is popularly known as Graph Attention Network (GAT: Veličković et al., 2018). This network assumes that the contributions of neighboring vertices to the central vertex are not pre-determined which is the case in the Graph Convolutional Network (Kipf & Welling, 2017). This adopts attention mechanisms to learn the relative weights between two connected vertices. The graph convolutional operation at iteration k is thereby defined as:

$$h_u^{(k)} = \sigma\left(\sum_{v\in\mathcal{N}(u)\cup u} \alpha_{uv}^{(k)}\Theta^{(k)}h_u^{(k-1)}\right) \tag{3}$$

where, h_u^k denotes the vertex-representation of a vertex u at iteration k; $h_u^{(0)} = X_u$ (the initial feature-vector associated with a vertex u). The connective strength between the vertex u and its neighbor vertex v is called attention weight, which is defined as

$$\alpha_{uv}^{(k)} = \mathrm{softmax}\left(\mathrm{LeakyReLU}\left(a^{\mathsf{T}}\left[\Theta^{(k)}h_u^{(k-1)} \parallel \Theta^{(k)}h_v^{(k-1)}\right]\right)\right) \tag{4}$$

where, a is the set of learnable parameters of a single layer feed-forward neural network, $\parallel$ denotes the concatenation operation.

Variant 4: GraphSAGE

This graph convolution is based on inductive representation learning on large graphs (Hamilton et al., 2017), which is primarily used to generate low-dimensional vector

representations for vertices. It adopts two steps: First, it samples a neighbourhood vertices of a vertex; Second, aggregate the feature-information from these sampled vertices. GraphSAGE is used to found to be very useful for graphs with vertices associated with rich feature-vectors. The following is an iterative update of the vertex representations in a graph:

$$h_u^{(k)} = \sigma\left(h_u^{(k-1)} \cdot \Theta_1^{(k)} + \frac{1}{|\mathcal{N}(u)|} \sum_{v \in \mathcal{N}(u)} h_v^{(k-1)} \cdot \Theta_2^{(k)} \right) \tag{5}$$

where, h_u^k denotes the vertex-representation of a vertex u at iteration k, σ is a non-linear transfer function applied component wise to the function argument, $\mathcal{N}$ denotes the neighborhood function, Θs are the layer-specific learnable parameters of the network.

Variant 5: ARMA

This graph convolution is inspired by the auto-regressive moving average (ARMA) filters that are considered to be more robust than polynomial filters (Bianchi et al., 2021). The ARMA graph convolutional operation is defined as:

$$\mathbf{H}^{(k)} = \frac{1}{M} \sum_{m=1}^{M} \mathbf{H}_m^{(K)} \tag{6}$$

where, $\mathbf{H}^k$ denotes the vertex-representation matrix at iteration k, M is the number of parallel stacks, K is the number of layers; and $\mathbf{H}_m^{(K)}$ is recursively defined as

$$\mathbf{H}_m^{(k+1)} = \sigma\left(\hat{L} \mathbf{H}_m^{(k)} \Theta_2^{(k)} + \mathbf{H}^{(0)} \Theta_2^{(k)} \right) \tag{7}$$

where, σ is a non-linear transfer function, $\hat{L} = I - L$ is the modified Laplacian. The Θ parameters are learnable parameters.

B.2 Graph pooling

Graph pooling methods apply the idea of downsampling mechanisms to graphs. This operation allows us to obtain refined graph representations at each layer. The primary aim of including a graph pooling operation after each graph convolution is that this operation can reduce the graph representation while ideally preserving important structural information. In this work, we use a recently proposed graph pooling method based on self-attention (Lee et al., 2019). This method uses the graph convolution defined in Eq. (1) to obtain a self-attention score as given in Eq. 8 with the trainable parameter replaced by $\Theta_{att} \in \mathbb{R}^{N \times 1}$, which is a set of trainable parameters in the pooling layer.

$$Z = \sigma\left(\tilde{D}^{-\frac{1}{2}} \tilde{A} \tilde{D}^{-\frac{1}{2}} \mathbf{X} \Theta_{att} \right) \tag{8}$$

Here, $\sigma(\cdot)$ is the activation function e.g. tanh.

B.3 Hierarchical graph pooling

The graph-convolution and graph-pooling operations described in the preceding two subsections allows a GNN are concerned with construction of vertex-representations. To deal

with the problem of graph classification (as is the case in this work), we need to represent an input graph as a "flattened" fixed-length feature-vector that can then be used with a standard fully-connected multilayer neural network (e.g. Multilayer Perceptron) to produce a class-label. To construct this graph-representation (mostly, a dense real-valued feature-vector, also called a *graph-embedding*), we use hierarchical graph-pooling method proposed by Cangea et al. (2018). This method is implemented with two operations: (a) global average pooling, that averages all the learnt vertex representations in the final (readout) layer; (b) augmenting the representation obtained in (a) with the representation obtained using global max pooling, that seek to obtained the most relevant information and could strengthen the graph-representation. The term "hierarchical" refers to the fact that the above two operations (a) and (b) are carried out after each conv-pool block in the GNN (refer Fig. 4). The final graph representation is an aggregate of all the layer-wise representations by taking their sum.

The output graph after each conv-pool block can be represented by a concatenation of the global average pool representation and the global max pool representation as:

$$H_G^{(k)} = \text{avg}(\mathbf{H}^{(k)}) \,||\, \max(\mathbf{H}^{(k)}) \tag{9}$$

where, H_G^k denotes the graph-representation at iteration k; $\mathbf{H}^{(k)}$ denotes the matrix of vertex-representations after conv-pool operations at iteration k as mathematically described in Sects. B.1 and B.2; avg and max denote the average and max operations, which are computed as follows:

$$\text{avg}(\mathbf{H}^{(k)}) = \frac{1}{N} \sum_{i=1}^{N} \mathbf{H}_i^{(k)} \tag{10}$$

$$\max(\mathbf{H}^{(k)}) = \max_{i=1}^{N} \mathbf{H}_i^{(k)} \tag{11}$$

Here, $\mathbf{H}_i^k$ denotes the representation for the ith vertex of the graph; N is the number of nodes in the graph.

The final fixed-length representation after iteration K for the whole input graph is then computed by the element-wise sum, denoted as $\oplus$, of these intermediate graph-representations in Eq. 9:

$$H_G^{(K)} = \oplus_{k=1}^{K} H_G^{(k)} \tag{12}$$

In our present work, $K = 3$ (since, we use 3 conv-pool blocks in our GNN). The graph-representation $H_G^{(K)}$ is then input to a multilayer perceptron as described in Sect. 5.3.

Appendix C: Application details

C.1 Mode-declarations

We use the ILP engine, Aleph (Srinivasan, 2001) to construct the most-specific clause for a relational data instance given background-knowledge, mode specifications and a depth. The mode-language used for our main experiments in the paper is given below:

```
:- modeb(*,bond(+mol,-atomid,-atomid,#atomtype,#atomtype,#bondtype)).
:- modeb(*,has_struc(+mol,-atomids,-length,#structype)).
:- modeb(*,connected(+mol,+atomids,+atomids)).
:- modeb(*,fused(+mol,+atomids,+atomids)).
```

The '#'-ed arguments in the mode declaration refers to type, that is, `#atomtype` refers to the type of atom, `#bondtype` refers to the type of bond, and `#structype` refers to the type of the structure (functional group or ring) associated with the molecule.

C.2 Logical representation of the data and background knowledge

Data in the primary set of experiments are molecules. At the lowest level, the atoms and bonds in each molecule are represented as a set of ground definitions of the `bond/6` predicate. Thus `bond(m1,27,24,o2,car,1)` denotes that in instance `m1` there is an oxygen atom (id 27), and a carbon atom (id 24) connected by a single bond (`car` denotes a carbon atom in an aromatic ring). Definitions of functional-groups and ring-structures are written as clausal definitions that use the definitions of this low-level `bond/6` predicate. There are about 100 such higher-level definitions. The result of inference about the presence of functional groups and rings is pre-compiled for efficiency. For example:

```
functional_group(m1,[27],1,oxide).
ring(m1,[25,28,30,29,26,23],6,benzene_ring).
```

Here, the 1 and 6 denote number of atoms involved in the group or ring. Access to groups and rings in a molecule uses the `has_struc/4` predicate:

```
has_struc(Mol,Atoms,Length,Type):-
    ring(Mol,Atoms,Length,Type).
has_struc(Mol,Atoms,Length,Type):-
    functional_group(Mol,Atoms,Length,Type).
...
```

In addition to functional groups and rings, there are predicates for computing if structures are connected or fused. An example of the most-specific clause for a data instance is:

```
class(m411,pos):-
    bond(m411,18,13,c3,o2,1),
    bond(m411,13,18,o2,c3,1),
    bond(m411,12,16,nar,nar,ar),...,
    has_struc(m411,[5,7,10,6,4],5,pyrrole_ring),
    has_struc(m411,[7,11,16,12,8,5],6,pyridazine_ring),...,
    connected(m411,[15],[9,14,13]),
    connected(m411,[15],[7,11,16,12,8,5]),...,
    fused(m411,[7,11,16,12,8,5],[5,7,10,6,4]),...,
    lteq(1,1),lteq(3,3),...,
    gteq(1,1),gteq(3,3),....
```

C.3 Propositionalisation experiments

In a propositionalisation approach (Lavrač et al., 1991), each data instance is represented using a Boolean vector of 0's and 1's, depending on the value of propositions (constructed manually or automatically) for the data instance (the value of the i^{th} dimension is 0 if the i^{th} proposition is false for the data-instance and 1 otherwise). The resulting dataset is then used to construct an MLP model. The following details are relevant:

- The MLP is implemented using Tensorflow-Keras (Chollet et al., 2015).
- The number of layers in MLP is tuned using a validation-based approach. The parameter grid for number of hidden layers is: $\{1, 2, 3, 4\}$.
- Each layer has fixed number of neurons: 10.
- The dropout rate is 0.5. We apply dropout (Srivastava et al., 2014) after every layer in the network except the output layer.
- The activation function used in each hidden layer is `relu`.
- The training is carried out using the Adam optimiser (Kingma & Ba, 2015) with learning rate 0.001.
- Additionally, we use early-stopping (Prechelt, 1998) to control over-fitting during training.

For the DRM, propositions simply denote whether any specific relation in the background knowledge is true or false for the data instance. BCP (França et al., 2014) constructs propositions using the most-specific clauses returned by the ILP system Aleph given the background-knowledge B, modes M and depth-limit d. For the construction of Boolean features using BCP, we use the code available at (Jankovics, 2020).

C.4 Experiments with ILP benchmarks

The seven datasets are taken from (Srinivasan et al., 2003). These datasets are some of the most popular benchmark datasets to evaluate various techniques within ILP studies. For the construction of BotGNNs the following details are relevant:

- There is a BK for each dataset.
- There are 10 splits for each dataset. Therefore, for each test-split we construct BotGNNs (all 5 variants), using 8 of rest splits as training set and the remaining 1 split as a validation set.
- Since these datasets are small (few hundreds of data instances), we could manage to perform some hyperparameter tuning for construction of our BotGNNs. The parameter grids for this are: $m : \{8, 16, 32, 64, 128\}$; batch-size: $\{16, 32\}$; learning rate: $\{0.0001, 0.0005, 0.001\}$.
- Other details are same as described in the main BotGNN experiments.
- In Sect. 5.5 we report the test accuracy from the best performing BotGNN variant.

Acknowledgements A.S. is a Visiting Professorial Fellow at School of CSE, UNSW Sydney. He is also the Class of 1981 Chair Professor at BITS Pilani. We thank Gustav Šourek, Czech Technical University, Prague for providing the dataset information; and researchers at the DTAI, University of Leuven, for suggestions on how to use the background knowledge within DMAX. We also thank Oghenejokpeme I. Orhobor and Ross D. King for providing us with the initial set of background-knowledge definitions. We thank Artur d'Avila Garcez, City, University of London for providing information on the usage of BCP within their system CILP++.

Data and Code Availability Data and codes used in our experiments are available at: https://github.com/tirtharajdash/BotGNN.

References

Ando, H. Y., Dehaspe, L., Luyten, W., Van Craenenbroeck, E., Vandecasteele, H., & Van Meervelt, L. (2006). Discovering h-bonding rules in crystals with inductive logic programming. *Molecular Pharmaceutics, 3*(6), 665–674.

Bai, S., Zhang, F., & Torr, P. H. (2021). Hypergraph convolution and hypergraph attention. *Pattern Recognition, 110*, 107637.

Besold, T. R., Garcez, A. D., Bader, S., Bowman, H., Domingos, P., Hitzler, P., Kühnberger, K. U., Lamb, L. C., Lowd, D., Lima, P. M. V., et al. (2017). Neural-symbolic learning and reasoning: A survey and interpretation. arXiv:abs/1711.03902

Bianchi, F. M., Grattarola, D., Livi, L., & Alippi, C. (2021). Graph neural networks with convolutional arma filters. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, pp. 1–1. https://doi.org/10.1109/TPAMI.2021.3054830

Bravo, H. C., Page, D., Ramakrishnan, R., Shavlik, J., & Costa, V. S. (2005). A framework for set-oriented computation in inductive logic programming and its application in generalizing inverse entailment. In *International conference on inductive logic programming*, Springer, pp. 69–86.

Cangea, C., Veličković, P., Jovanović, N., Kipf, T., & Liò, P. (2018). Towards sparse hierarchical graph classifiers. arXiv:abs/1811.01287

Chollet, F., et al. (2015). Keras. https://keras.io.

Dash, T., Srinivasan, A., Vig, L., Orhobor, O. I., & King, R. D. (2018). Large-scale assessment of deep relational machines. In *International conference on inductive logic programming*, Springer, pp. 22–37.

Dash, T., Srinivasan, A., Joshi, R. S., & Baskar, A. (2019). Discrete stochastic search and its application to feature-selection for deep relational machines. In *International conference on artificial neural networks*, Springer, pp. 29–45.

Dash, T., Chitlangia, S., Ahuja, A., & Srinivasan, A. (2021a). How to tell deep neural networks whatwe know. arXiv:abs/2107.10295

Dash, T., Chitlangia, S., Ahuja, A., & Srinivasan, A. (2021b). Incorporating domain knowledge into deep neural networks. arXiv:abs/2103.00180

Dash, T., Srinivasan, A., & Vig, L. (2021c). Incorporating symbolic domain knowledge into graph neural networks. *Machine Learning*, 1–28.

Du, S. S., Hou, K., Salakhutdinov, R. R., Poczos, B., Wang, R., & Xu, K. (2019). Graph neural tangent kernel: Fusing graph neural networks with graph kernels. *Advances in Neural Information Processing Systems, 32*, 5723–5733.

Feng, Y., You, H., Zhang, Z., Ji, R., & Gao, Y. (2019). Hypergraph neural networks. *Proceedings of the AAAI Conference on Artificial Intelligence, 33*, 3558–3565. https://doi.org/10.1609/aaai.v33i01.33013558

Fey, M., & Lenssen, J. E. (2019). Fast graph representation learning with PyTorch Geometric. In *ICLR workshop on representation learning on graphs and manifolds*.

Fischer, M., Balunovic, M., Drachsler-Cohen, D., Gehr, T., Zhang, C., & Vechev, M. (2019). Dl2: Training and querying neural networks with logic. In *International conference on machine learning*, PMLR, pp. 1931–1941.

França, M. V., Zaverucha, G., & Garcez, ASd. (2014). Fast relational learning using bottom clause propositionalization with artificial neural networks. *Machine Learning, 94*(1), 81–104.

Frasconi, P., Costa, F., De Raedt, L., & De Grave, K. (2014). klog: A language for logical and relational learning with kernels. *Artificial Intelligence, 217*, 117–143.

Garcez, A. D., Gori, M., Lamb, L. C., Serafini, L., Spranger, M., & Tran, S. N. (2019). Neural-symbolic computing: An effective methodology for principled integration of machine learning and reasoning. *FLAP, 6*(4), 611–632.

Garcez, A. S. A., & Zaverucha, G. (1999). The connectionist inductive learning and logic programming system. *Applied Intelligence, 11*(1), 59–77.

Gilmer, J., Schoenholz, S. S., Riley, P. F., Vinyals, O., & Dahl, G. E. (2017). Neural message passing for quantum chemistry. In *International conference on machine learning*, PMLR, pp. 1263–1272.

Gori, M., Monfardini, G., & Scarselli, F. (2005). A new model for learning in graph domains. In *Proceedings. 2005 IEEE international joint conference on neural networks, 2005*, IEEE, vol. 2, pp. 729–734.

Hamilton, W., Ying, Z., & Leskovec, J. (2017). Inductive representation learning on large graphs. In *Advances in neural information processing systems*, pp. 1024–1034.

Hamilton, W. L. (2020). Graph representation learning. *Synthesis Lectures on Artifical Intelligence and Machine Learning, 14*(3), 1–159.

Heckerman, D., Meek, C., & Koller, D. (2007). Probabilistic entity-relationship models, prms, and plate models. *Introduction to statistical relational learning*, pp. 201–238.

Jankovics, V. (2020). vakker/cilp. https://github.com/vakker/CILP

Johnson, J., Hariharan, B., van der Maaten, L., Fei-Fei, L., Lawrence Zitnick, C., & Girshick, R. (2017). Clevr: A diagnostic dataset for compositional language and elementary visual reasoning. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 2901–2910.

Kersting, K., Kriege, N. M., Morris, C., Mutzel, P., & Neumann, M. (2016). Benchmark data sets for graph kernels. http://graphkernels.cs.tu-dortmund.de

King, R. D., Rowland, J., Oliver, S. G., Young, M., Aubrey, W., Byrne, E., Liakata, M., Markham, M., Pir, P., Soldatova, L. N., et al. (2009). The automation of science. *Science, 324*(5923), 85–89.

Kingma, D. P., & Ba, J. (2015). Adam: A method for stochastic optimization. In *ICLR (Poster)*, arXiv:1412.6980

Kipf, T. N., & Welling, M. (2017). Semi-supervised classification with graph convolutional networks. In *5th International conference on learning representations, ICLR 2017*, Toulon, France, April 24–26, 2017, Conference Track Proceedings.

Kitano, H. (2016). Artificial intelligence to win the nobel prize and beyond: Creating the engine for scientific discovery. *AI Magazine, 37*(1), 39–49.

Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2017). Imagenet classification with deep convolutional neural networks. *Communications of the ACM, 60*(6), 84–90.

Kursuncu, U., Gaur, M., & Sheth, A. (2020). Knowledge infused learning (k-il): Towards deep incorporation of knowledge in deep learning. arXiv:abs/1912.00512

Lamb, L. C., Garcez, A. D., Gori, M., Prates, M. O., Avelar, P. H., & Vardi, M. Y. (2020). Graph neural networks meet neural-symbolic computing: A survey and perspective. In C. Bessiere (Ed.) *Proceedings of the twenty-ninth international joint conference on artificial intelligence, IJCAI-20, international joint conferences on artificial intelligence organization*, pp. 4877–4884. https://doi.org/10.24963/ijcai.2020/679, survey track.

Lavrač, N., Džeroski, S., & Grobelnik, M. (1991). Learning nonrecursive definitions of relations with linus. In *European working session on learning*, Springer, pp. 265–281.

Lavrač, N., Podpečan, V., & Robnik-Šikonja, M. (2021). *Propositionalization of relational data* (pp. 83–105). Cham: Springer. https://doi.org/10.1007/978-3-030-68817-2_4

Lee, J., Lee, I., & Kang, J. (2019). Self-attention graph pooling. In *International conference on machine learning*, pp. 3734–3743.

Lodhi, H. (2013). Deep relational machines. In *International conference on neural information processing*, Springer, Berlin, pp. 212–219.
Marx, K. A., O'Neil, P., Hoffman, P., & Ujwal, M. (2003). Data mining the nci cancer cell line compound gi50 values: identifying quinone subtypes effective against melanoma and leukemia cell classes. *Journal of Chemical Information and Computer Sciences*, *43*(5), 1652–1667.
Morris, C., Ritzert, M., Fey, M., Hamilton, W. L., Lenssen, J. E., Rattan, G., & Grohe, M. (2019). Weisfeiler and leman go neural: Higher-order graph neural networks. *Proceedings of the AAAI Conference on Artificial Intelligence, 33*, 4602–4609.
Muggleton, S. (1995). Inverse entailment and progol. *New Generation Computing, 13*(3–4), 245–286.
Muggleton, S., & De Raedt, L. (1994). Inductive logic programming: Theory and methods. *The Journal of Logic Programming, 19*, 629–679.
Muralidhar, N., Islam, M. R., Marwah, M., Karpatne, A., & Ramakrishnan, N. (2018). Incorporating prior domain knowledge into deep neural networks. In *2018 IEEE International conference on big data (Big Data)*, IEEE, pp. 36–45.
Oord, A. V. D., Dieleman, S., Zen, H., Simonyan, K., Vinyals, O., Graves, A., Kalchbrenner, N., Senior, A., & Kavukcuoglu, K. (2016). Wavenet: A generative model for raw audio. arXiv:abs/1609.03499
Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., et al. (2019). Pytorch: An imperative style, high-performance deep learning library. In *Advances in neural information processing systems*, pp. 8024–8035.
Plotkin, G. (1972). *Automatic methods of inductive inference*. The University of Edinburgh. Ph.D. dissertation.
Plotkin, G. D. (1970). A note on inductive generalization. *Machine intelligence, 5*(1), 153–163.
Prechelt, L. (1998). Early stopping-but when? In *Neural Networks: Tricks of the trade*, Springer, Berlin, pp. 55–69.
Saha, A., Srinivasan, A., & Ramakrishnan, G. (2012). What kinds of relational features are useful for statistical learning? In *International conference on inductive logic programming*, Springer, pp. 209–224.
Schlichtkrull, M., Kipf, T.N., Bloem, P., Van Den Berg, R., Titov, I., & Welling, M. (2018). Modeling relational data with graph convolutional networks. In *European semantic web conference*, Springer, pp. 593–607.
Sheth, A., Gaur, M., Kursuncu, U., & Wickramarachchi, R. (2019). Shades of knowledge-infused learning for enhancing deep learning. *IEEE Internet Computing, 23*(6), 54–63.
Sourek, G., Aschenbrenner, V., Zelezny, F., Schockaert, S., & Kuzelka, O. (2018). Lifted relational neural networks: Efficient learning of latent relational structures. *Journal of Artificial Intelligence Research, 62*, 69–100.
Šourek, G., Železnỳ, F., & Kuželka, O. (2021). Beyond graph neural networks with lifted relational neural networks. *Machine Learning*, pp. 1–44.
Srinivasan, A. (2001). The aleph manual. https://www.cs.ox.ac.uk/activities/programinduction/Aleph/aleph.html
Srinivasan, A., & Ramakrishnan, G. (2011). Parameter screening and optimisation for ilp using designed experiments. *Journal of Machine Learning Research*, *12*(2).
Srinivasan, A., King, R. D., & Bain, M. E. (2003). An empirical study of the use of relevance information in inductive logic programming. *Journal of Machine Learning Research*, *4*(Jul):369–383.
Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., & Salakhutdinov, R. (2014). Dropout: A simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research, 15*(1), 1929–1958.
Stevens, R., Taylor, V., Nichols, J., Maccabe, A. B., Yelick, K., & Brown, D. (2020). Ai for science. Tech. rep., Argonne National Lab.(ANL), Argonne, IL (USA).
Van Craenenbroeck, E., Vandecasteele, H., & Dehaspe, L. (2002). Dmax's functional group and ring library. https://dtai.cs.kuleuven.be/software/dmax/
Veličković, P., Cucurull, G., Casanova, A., Romero, A., Liò, P., & Bengio, Y. (2018). Graph attention networks. In *International conference on learning representations*, https://openreview.net/forum?id=rJXMpikCZ
Vishwanathan, S. V. N., Schraudolph, N. N., Kondor, R., & Borgwardt, K. M. (2010). Graph kernels. *Journal of Machine Learning Research, 11*, 1201–1242.
Wang, H., Zhao, M., Xie, X., Li, W., & Guo, M. (2019). Knowledge graph convolutional networks for recommender systems. In *The world wide web conference*, pp. 3307–3313.
Williams, K., Bilsland, E., Sparkes, A., Aubrey, W., Young, M., Soldatova, L. N., De Grave, K., Ramon, J., De Clare, M., Sirawaraporn, W., et al. (2015). Cheaper faster drug development validated by the repositioning of drugs against neglected tropical diseases. *Journal of the Royal society Interface, 12*(104), 20141289.

Wu, Y., Schuster, M., Chen, Z., Le, Q. V., Norouzi, M., Macherey, W., Krikun, M., Cao, Y., Gao, Q., Macherey, K., et al. (2016). Google's neural machine translation system: Bridging the gap between human and machine translation. arXiv:abs/1609.08144

Wu, Z., Pan, S., Chen, F., Long, G., Zhang, C., & Philip, S. Y. (2020). A comprehensive survey on graph neural networks. *IEEE Transactions on Neural Networks and Learning Systems*.

Xie, Y., Xu, Z., Kankanhalli, M. S., Meel, K. S., & Soh, H. (2019). Embedding symbolic knowledge into deep networks. In *Advances in neural information processing systems*, pp. 4233–4243.

Xu, J., Zhang, Z., Friedman, T., Liang, Y., & Broeck, G. (2018). A semantic loss function for deep learning with symbolic knowledge. In *International conference on machine learning*, PMLR, pp. 5502–5511.

Xu, K., Hu, W., Leskovec, J., & Jegelka, S. (2019). How powerful are graph neural networks? In *International conference on learning representations*, https://openreview.net/forum?id=ryGs6iA5Km