Chapter 5

Complete Inclusion of Relational Information using Inverse Entailment*

At the end of the previous chapter, we showed that using vertex-enrichment as a technique for inclusion of domain-knowledge results in a loss of information. For example, a vertex in a vertex-enriched graph does not encode information that it is a part of multiple relations with the same predicate symbol and arity. We therefore say vertex-enrichment is a *simplified* approach to inclusion of relational information. In this chapter, we attempt to rectify this deficiency by including all of the relational information available into a graph-based neural network or GNN. The technique we propose is based on inverse entailment developed in the Inductive Logic Programming (ILP) literature. Here, for any given relational data instance, the inverse-entailment technique allows us to identify all the domain-relations entailed by the domain-knowledge via the construction of a *bottom-clause*. We propose a method to represent a bottom-clause using a *bottom-graph* that can be converted into a form suitable for GNN implementations. Overall, this transformation from a bottom-clause to a bottom-graph allows a principled way for complete inclusion of domain-knowledge into GNNs: we use the term "BotGNNs" for this form of graph neural networks. We evaluate the use of domain-knowledge in this manner using the relational datasets and background knowledge studied in the previous chapters. The main contributions of this chapter are as follows: (1) This chapter proposes a systematic technique using the method of inverse entailment in ILP, in particular, Mode-Directed Inverse-Entailment (MDIE) for complete inclusion of symbolic domain-knowledge into GNNs. The proposal in this chapter is a new technique for neuro-symbolic learning, called Bottom-Graph Neural Networks or BotGNNs; (2) This chapter provides a large-

*The content of this chapter is based on the following:
T. Dash, A. Srinivasan, A. Baskar, "Inclusion of domain-knowledge into GNNs using mode-directed inverse entailment", *Machine Learning*, 2021; https://doi.org/10.1007/s10994-021-06090-8.

scale evaluation of BotGNNs, constructed using relational data and domain-knowledge.

5.1 Mode-Directed Inverse Entailment

Mode-directed Inverse Entailment (MDIE) was introduced by Muggleton in [Mug95], as a technique for constraining the search for explanations for data in Inductive Logic Programming (ILP). For this chapter, 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", such that

Prior Necessity. $B \not\models E^+$

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

Weak Posterior Sufficiency. For every $D_j \in H$, $B \cup \{D_j\} \models p_1 \vee p_2 \vee \cdots \vee p_N$

Strong Posterior Sufficiency. $B \cup H \models E^+$

Posterior Satisfiability. $B \cup H \cup E^- \not\models \square$

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 [Mug95] is to constrain the identification of D_j using a *most-specific clause*. The following is adapted from [Mug95].

Remark 5.1 (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 $\neg a_1 \vee \neg a_2 \vee \cdots \vee \neg a_n$. Furthermore, since the a_is' are ground, $\bot_B(e)$ is a ground clause.*

[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 $\mathcal{C} = D_1 \wedge D_2 \wedge \cdots \wedge D_k$.

[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.

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 5.1. *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 comma (",") 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:

$$\neg gparent(henry, john) \wedge father(henry, jane) \wedge mother(jane, john) \wedge mother(jane, alice) \wedge parent(henry, jane) \wedge parent(jane, alice) \wedge parent(jane, john)$$

$\bot_B(e) = \overline{A}$:

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

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:

$$\{gparent(henry, john), \neg father(henry, jane), \ldots, \neg parent(jane, alice)\}.$$

$\bot_B(e)$ 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 chapter, 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 5.2. *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:

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

$$\vdots$$
$$methyl(m_1, [\ldots]),$$
$$\vdots$$

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.

5.1.1 Modes

Practical ILP systems like Progol [Mug95] use a *depth-bounded* bottom clause constructed within a mode-language. We first illustrate a simple example of a mode-language specification.

Example 5.3. *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*

$$\mathrm{M} = \{modeh(gparent(+person, -person)), modeb(father(+person, -person)),$$
$$modeb(mother(+person, -person)), modeb(parent(+person, -person))\}.$$

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 [Mug95], 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 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 5.4. *Continuing Example 5.3, in the following X, Y, Z are variables of type person. These clauses are all within the mode language specified in [Mug95]:*

(a) $gparent(X, Y) \leftarrow parent(X, Y)$;

[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.

(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 [Mug95]:

(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 [Mug95] for more details on the use of modes. Here we confine ourselves to the details necessary for the material in this chapter. We first reproduce the notion of a place-number of a term in a literal following [Plo72].

Definition 5.1 (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 5.5. *Let us look at the following examples:*
(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 5.2 (Mode-Declaration)**.** *The definition is as follows:*

(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 $mt'_k s$ 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(λ') or modeb(λ'). 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.

5.1.2 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

[5] For all experiments in this chapter, 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 [Mug95], this is usually provided as part of the mode declaration.

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 5.6. *Using the modes* M *in Example 5.3, we obtain the following most-specific clauses for the gparent example (Example 5.1):*

$\bot_{B,\mathrm{M},1}(e)$:

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

$\bot_{B,\mathrm{M},2}(e)$:

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

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

Definition 5.3 (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 5.4 (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 5.7. *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 Defn. 5.2), 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 chapter, we will mainly be interested in definite-clauses (that is, $m = 1$ in the definition that follows).

Definition 5.5 ($\lambda\mu$-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 $\lambda_1, \ldots, \lambda_m$ given $\mu_1, \ldots, \mu_m$; 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 γ 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 5.6 (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 5.8. *Let $\mathrm{M} = \{\mu_1, \mu_2, \mu_3, \mu_4, \mu_5, \mu_6\}$ where the mode declarations μ_is are as follows: $\mu_1 = modeh(p(+int))$, $\mu_2 = modeh(p(+real))$, $\mu_3 = modeb(q(+int))$, $\mu_4 = modeb(q(+real))$, $\mu_5 = modeb(r(+int))$, $\mu_6 = modeb(r(+real))$. 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 Defn. 5.6 does not allow the following two sequences to to be $\lambda\mu$-sequences: $\langle(\lambda_1, \mu_1), (\lambda_2, \mu_4), (\lambda_3, \mu_5)\rangle$ and $\langle(\lambda_1, \mu_2), (\lambda_2, \mu_4), (\lambda_3, \mu_5)\rangle$, because 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 [Mug95], clauses in $\mathcal{L}_{T_\Gamma,M,d}$ here are restricted to being ground (in [Mug95], clauses are required to have variables in $+$ and $-$ places of literals).

5.2 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 5.9. *Consider $\bot_{B,\mathrm{M},2}(e)$ in Example 5.6. The tabulation below shows the literals in $\bot_{B,\mathrm{M},2}(e)$ and matching modes.*

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 (τs) in literals appearing in $\bot_{B,\mathrm{M},2}(e)$ and their types (γs), obtained from the corresponding term-place number in the matching mode.

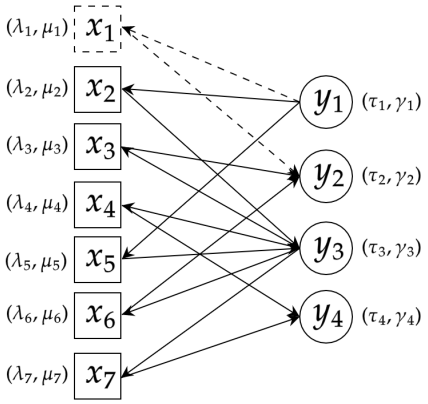
S.No.	Term (τ)	Type (γ)
1	$henry$	$person$
2	$john$	$person$
3	$jane$	$person$
4	$alice$	$person$

The information in these tables can be represented as a directed bipartite graph as shown in Figure 5.1. 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.

$\perp_{B,\mathrm{M},2}(e)$:

gparent(*henry*, *john*) ←
father(*henry*, *jane*),
mother(*jane*, *john*),
mother(*jane*, *alice*),
parent(*henry*, *jane*),
parent(*jane*, *john*),
parent(*jane*, *alice*)

(a)



(b)

Figure 5.1: 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.

The structure in Figure 5.1 is called a bottom-graph in this chapter. BotGNNs are GNN models constructed from graphs based on such clause-graphs. We first clarify some details needed for the construction of clause-graphs.

5.2.1 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;

[8]In this chapter, 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.

(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 $\Gamma' \cup \{\#t : t \in \Gamma' \text{ s.t. } \#t \text{ occurs in some mode } \mu \in \mathrm{M}\}$;

(h) LM to denote the set $\{(\lambda, \mu) : \lambda \in \Lambda, \mu \in \mathrm{M}, \mu \text{ is a mode-declaration for } \lambda\}$; and ET to denote the set $\{(\tau, \gamma) : \tau \in \mathrm{T}, \gamma \in \Gamma, \tau \text{ is of type } \gamma\}$;

(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 $\psi : (Xs \cup Ys) \to (LM \cup ET)$;

(l) We will use $CG_\top$ to denote the special graph $((\emptyset, \emptyset, \emptyset), \emptyset) \in \mathcal{G}$.

Functions. We assume bijections $h_x : LM \to Xs$; and $h_y : ET \to Ys$.

Implementation. We will assume the following implementation details:

1. The elements of Γ are assumed to be unary predicate symbols, and the type-definitions T_Γ in Defn. 5.3 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 Defn. 5.3) then $\gamma(\tau) \in B$. We will therefore refer to the mode-language $\mathcal{L}_{T_\Gamma,\mathrm{M},d}$ in Defn. 5.6 as $\mathcal{L}_{B,\mathrm{M},d}$;
2. An MDIE implementation that, given B, M, d, ensures for any ground definite-clause e returns a unique ground definite-clause $\bot_{B,\mathrm{M},d}(e) \in \mathcal{L}_{B,\mathrm{M},d}$ if it exists or $\emptyset$ otherwise. In addition, if $\bot_{B,\mathrm{M},d}(e) \in \mathcal{L}_{B,\mathrm{M},d}$, we assume the MDIE implementation has been extended to return at least one matching $\lambda\mu$-sequence for $\bot_{B,\mathrm{M},d}$.

5.2.2 Construction of Bottom-Graphs

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

Definition 5.7 (Literals Set)**.** *Given background knowledge B, a set of modes* M*, and 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 $Lits(C) = \emptyset$;

[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 $V = X \cup Y$.

(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 subsection 5.2.3).

Example 5.10. *We revisit the gparent example. Let* $\mathrm{M} = \{\mu_1, \mu_2, \mu_3, \mu_4\}$*, 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's assume that background knowledge B contains the type-definitions: $person(henry)$, $person(john)$, $person(jane)$, $person(alice)$; and the depth-bound be $d = 2$. Let $C = \bot_{B,\mathrm{M},d}(e)$ as in Example 5.6.

1. *Here C is the set consisting of the literals: $\{gparent(henry, john)$, $\neg father(henry, jane)$, $\neg mother(jane, john)$, $\neg mother(jane, alice)$, $\neg parent(henry, jane)$, $\neg parent(jane, john)$, $\neg parent(jane, alice)\}$.*

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 5.8 (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 5.11. *In Example 5.10, $Lits(C) = \{(\lambda_1, \mu_1), (\lambda_1, \mu_1), \ldots, (\lambda_7, \mu_4)\}$. Therefore, $Terms(C) = \{(\lambda_1, \mu_1, \langle 1\rangle), (\lambda_1, \mu_1, \langle 2\rangle), (\lambda_2, \mu_2, \langle 1\rangle), (\lambda_2, \mu_2, \langle 2\rangle), \ldots, (\lambda_7, \mu_4, \langle 1\rangle), (\lambda_7, \mu_4, \langle 2\rangle)\}$.*

Definition 5.9 (Clause-Graphs)**.** *Given background knowledge B, a set of modes* M*, and a depth-limit d, we define a function $ClauseToGraph : \mathcal{L}_{B,\mathrm{M},d} \to \mathcal{G}$ as follows.*

If $C = \emptyset$ then $ClauseToGraph(C) = CG_\top$ (see subsection 5.2.1). Otherwise, the clause-graph of C is given as $ClauseToGraph(C) = (G, \psi) \in \mathcal{G}$ where

- *The graph $G = (X, Y, E)$ is defined as:*
  - $X = \{x_i : (\lambda, \mu) \in Lits(C), x_i = h_x((\lambda, \mu))\}$;
  - $Y = \{y_j : (\lambda, \mu, \pi) \in Terms(C),\ TermType((\lambda, \mu, \pi)) = (\tau, \gamma),\ ModeType(\mu, \pi) \in \{(+, \gamma), (-, \gamma)\},\ y_j = h_y((\tau, \gamma))\} \cup \{y_j : (\lambda, \mu, \pi) \in Terms(C),\ TermType((\lambda, \mu, \pi)) = (\tau, \gamma),\ ModeType(\mu, \pi) = (\#, \gamma),\ y_j = h_y((\tau, \#\gamma))\}$;
  - $E = E_{in} \cup E_{out}$, *where:*

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

- *The vertex-labelling function ψ is defined as*

$$\psi(v) = \begin{cases} h_x^{-1}(v) & \text{if } v \in X, \\ h_y^{-1}(v) & \text{if } v \in Y. \end{cases}$$

In subsection 5.2.3, we show $ClauseToGraph(\cdot)$ is an injective function.

Example 5.12. *We continue Example 5.11. Recall $Terms(C) = \{(\lambda_1, \mu_1, \langle 1\rangle), (\lambda_1, \mu_1, \langle 2\rangle), (\lambda_2, \mu_2, \langle 1\rangle), \ldots, (\lambda_7, \mu_4, \langle 2\rangle)\}$. Then, in Defn. 5.9, the term-types are given as follows: $TermType((\lambda_1, \mu_1, \langle 1\rangle)) = (henry, person)$, $TermType((\lambda_1, \mu_1, \langle 2\rangle)) = (john, person)$, $TermType((\lambda_2, \mu_2, \langle 1\rangle)) = (henry, person)$, $\ldots$, $TermType((\lambda_7, \mu_4, \langle 2\rangle)) = (alice, person)$. Then $ClauseToGraph(C)$ is as follows:*

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

– $E = E_{in} \cup E_{out}$, *with*

$$E_{in} = \{(y_1, x_1), (y_1, x_2), (y_1, x_5), (y_3, x_3), (y_3, x_4), (y_3, x_6), (y_3, x_7)\}$$
$$E_{out} = \{(x_1, y_2), (x_2, y_3), (x_3, y_2), (x_4, y_4), (x_5, y_3), (x_6, y_2), (x_7, y_4)\}.$$

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

The reader can compare this to the graph shown diagrammatically in Figure 5.1.

Example 5.13. *Examples 5.10–5.12 do not illustrate what happens when we have multiple matching mode-declarations. To illustrate this we repeat the exercise with Example 5.8 (for consistency, we now use the symbol* $\mathbb{R}$ *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 5.8).*
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* $Terms(C) = \{(\lambda_1, \mu_1, \langle 1 \rangle), (\lambda_2, \mu_3, \langle 1 \rangle), \ldots, (\lambda_3, \mu_6, \langle 1 \rangle)\}$
7. *Then, in Defn. 5.9,* $TermType((\lambda_1, \mu_1, \langle 1 \rangle)) = (1, int)$, $TermType((\lambda_2, \mu_3, \langle 1 \rangle)) = (1, int)$, $\ldots$, $TermType((\lambda_3, \mu_6, \langle 1 \rangle)) = (1, \mathbb{R})$.

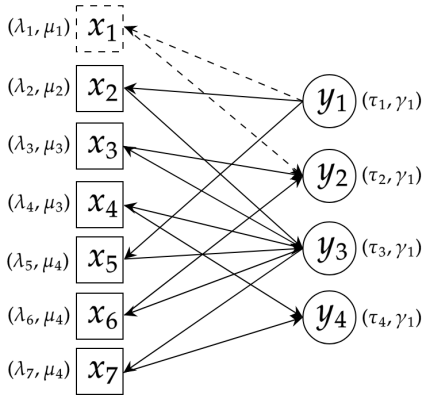
The reader can verify that $ClauseToGraph(C) = (G, \cdot)$ *where* $G = (X, Y, E)$ *s.t.*

- $X = \{x_1, x_2, \ldots, x_6\}$, *with* $x_1 = h_x((\lambda_1, \mu_1))$, $x_2 = h_x((\lambda_2, \mu_3))$,$\ldots$, $x_6 = h_x((\lambda_3, \mu_6))$
- $Y = \{y_1, y_2\}$, *with* $y_1 = h_y((1, int))$ *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 5.10 (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* $BotGraph_{B,\mathrm{M},d}(e) \equiv BotGraph : \mathcal{E} \to \mathcal{G}$ *as* $BotGraph(e) = ClauseToGraph(\bot_{B,\mathrm{M},d}(e))$.

Example 5.14. *For our gparent/2 example described through out this chapter, 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 following diagram. In the diagram, the "dashed" square-boxes and the "dashed" arrows are shown to indicate the vertices specifying the head of the clause:*



The vertex-labelling in the above graph is as obtained in Example 5.12, where γ_1 denotes the type-name person, τ_1,τ_2, τ_3, τ_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 Figure 5.1.

We now provide some properties of clause-graphs.

5.2.3 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 subsection 5.2.1). 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 $C = \{l_1, \ldots, l_m, \neg l_{m+1}, \ldots, \neg l_k\}$, where $1 \leq m < k$, we have Λ_C is the set $\{l_1, \ldots, l_{m+1}, \ldots, l_k\}$.

Definition 5.11 ($\preceq_{cg}$). *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 the following hold: (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 5.1. *$\langle \mathcal{G}, \preceq_{cg} \rangle$ is partially ordered.*

Proof: To prove this, 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 5.2. *The following are the consequences of the Defns. 5.7–5.9, and Defn. 5.11:*

(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, $\psi_1 \subseteq \psi_2$. Hence $CG_1 \preceq_{cg} CG_2$;

(ii) Let $C, D \in \mathcal{L}_{B,\mathrm{M},d}$. The clause-graphs of C and D are $CG_1 = ClauseToGraph(C) = ((X_1, Y_1, E_1), \psi_1)$ and $CG_2 = ClauseToGraph(D) = ((X_2, Y_2, E_2), \psi_2)$, respectively. 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 $X_1 \subseteq X_2$, which further combines with the observation (i) above, we get $CG_1 \preceq_{cg} CG_2$.

Lemma 5.1 (*Lits*)**.** *The function $Lits : \mathcal{L}_{B,M,d} \to 2^{LM}$ (defined in Defn. 5.7) 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 because $Lits(C)$ and $Lits(D)$ contain all $\lambda\mu$-sequences for C, D. ■

Lemma 5.2. *Let $C, D \in \mathcal{L}_{B,\mathrm{M},d}$. Let the corresponding clause-graphs of these two clauses be $CG_1 = ClauseToGraph(C)$ and $CG_2 = ClauseToGraph(D)$. if $C = D$ then $CG_1 = CG_2$.*

Proof: The result holds trivially if $C = D = \emptyset$; therefore 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 5.1 $Lits(C) = Lits(D)$. From Defn. 5.8, $Terms(C) = Terms(D)$ iff $Lits(C) = Lits(D)$. From Defn. 5.9, $X_1 = X_2$ 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, $\psi_1 = \psi_2$. This implies, $CG_1 = CG_2$. ■

Proposition 5.2 (*ClauseToGraph*)**.** *The function* $ClauseToGraph : \mathcal{L}_{B,\mathrm{M},d} \rightarrow \mathcal{G}$ *(defined in Defn. 5.9) is injective.*

Proof: Let C and D in $\mathcal{L}_{B,\mathrm{M},d}$, and the corresponding clause-graphs of these two clauses be $CG_1 = ClauseToGraph(C)$ 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, because $X_1 = X_2$. Similarly, we can prove for $\lambda_i \in D$ and $\lambda_i \notin C$. ■

Proposition 5.3 (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 $x_i \in X$, consider the following sets:

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

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 (Defn. 5.5) for C s.t. some $(\lambda_i, \mu_i) \in Lits(C)$ (Defn. 5.7) and $h_x((\lambda_i, \mu_i)) \in X$ (Defn. 5.9). 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 hence $C = C'$. ■

Remark 5.3. *We note the following without formal proofs:*

(i) In Defn. 5.10, 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 Defn. 5.12, $Antecededent : \mathcal{G} \to \mathcal{G}$ *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, it is possible that* $CG_1 \neq CG_2$, *and* $Antecedent(CG_1) = Antecedent(CG_2)$.

(iii) In Defn. 5.13, $UGraph : \mathcal{G} \to \mathcal{G}$ *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* $UGraph(CG_1) = UGraph(CG_2)$*).*

Remark 5.4 (Size of a Bottom-Graph)**.** *We find the bound on the size of a clause-graph obtained from a most-specific clause–the bottom-graph–by using the result in [Mug95, Theorem 26].*

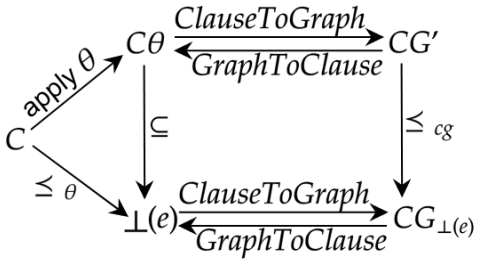
Let $\bot_{B,\mathrm{M},d}(e)$ *denote a most-specific clause for an example* e, *given* B, M, d, *and* $((X, Y, E), \psi)$ *denote the corresponding clause-graph. Let* j^+ *denote an upper-bound on the number of* $+$ *arguments in modeb declarations in* M *and the number of* $-$,# *arguments in modeh declarations in* M. *Let* j^- *denote an upper-bound on the number of* $-$,# *arguments in modeb declarations in* M *and the number of* $+$ *arguments in modeh declarations in* M. *Then the size of* $\bot_{B,\mathrm{M},d}$ *is bounded by* $(r|\mathrm{M}|j^+j^-)^{dj^+}$, *where* r *is a constant called the "recall" number (see [Mug95] for details). For each literal in* $\bot_{B,\mathrm{M},d}$ *there is 1 vertex in* X. *Also, for every argument in each literal in* $\bot_{B,\mathrm{M},d}$ *there is a vertex in* Y. *It is straightforward to see that the size of the corresponding clause-graph is bounded by* $(r|\mathrm{M}|j^+j^-)^{dj^+}(1 + j^+ + j^-)$.

Finally, we relate the clausal explanations found by some MDIE systems using the ordering $\preceq_\theta$ defined over clauses in [Plo70].[11] 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 5.5 (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:*

[11] $C_1 \preceq_\theta C_2$ 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*.*

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.

5.2.4 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 chapter. 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 5.12 (Antecedent-Graphs)**.** *We define function* $Antecedent : \mathcal{G} \to \mathcal{G}$ *as follows. Let* $(G, \psi) \in \mathcal{G}$*, where* $G = (X, Y, E)$ *is a directed bipartite graph. Let* $X_h = \{x \in X : \psi(x) = (\lambda, \mu) with\ \mu = modeh(\cdot)\}$*. We define* (G', ψ') *where* $G' = (X', Y', E')$ *such that*

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

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

Most GNN implementations, including those used in this chapter, require graphs to be undirected [Ham20]. 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 as follows:

Definition 5.13 (Undirected Clause-Graphs)**.** *We define a function* $\mathit{UGraph} : \mathcal{G} \to \mathcal{G}$ *as follows. Let* $(G, \psi) \in \mathcal{G}$, *where* $G = (X, Y, E)$ *is a directed bipartite graph. We define* (G', ψ'), *where* $G' = (X', Y', E')$ *such that*

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

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

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 5.14 (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* $\mu \in \mathrm{M}$. *Let* $\mathrm{T}_\# \subseteq \mathrm{T}$ *be the set of ground-terms of types in* $\Gamma_\#$.[12]

For $v \in X \cup Y$, *let us define the following four functions from* $X \cup Y$ *to the set of all*

[12] 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.

real vectors of finite length:

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

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

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

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

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 $x \in S$.[13]

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 Figure 4.1) in section 4.1. 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 5.15. *Recall the most-specific clause for the $gparent(henry, john)$ in Example 5.1:*

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

The clause-graph and corresponding antecedent-graph are shown below.
Assume the following sets:

$$P = \{gparent/2, father/2, mother/2, parent/2\},$$
$$\Gamma = \{person\},$$
$$\Gamma_\# = \emptyset.$$

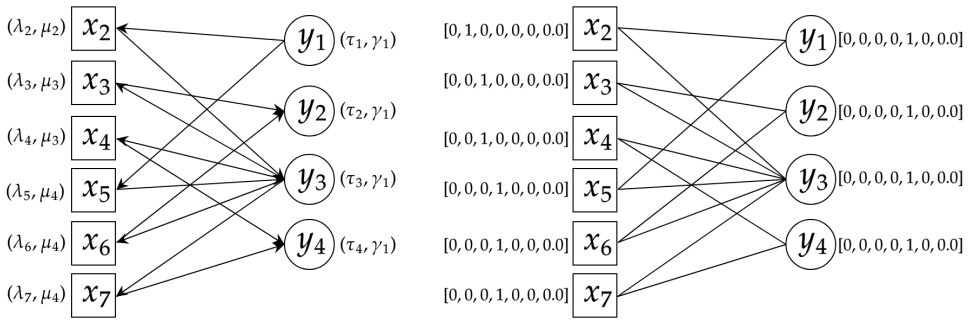
Additionally, since the mode-language in Example 5.1 does not have any #'ed arguments, $\mathrm{T}_\# = \emptyset$. Therefore, f_ρ is a 4-dimensional (one-hot encoded) vector (since $|P| = 4$); f_γ

[13] 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.

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 5.12; 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)	$[0,0,0,0]$	$[1]$	$[0]$	$[0.0]$	$[0,0,0,0,1,0,0.0]$

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 BotGNN-Graph, 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 5.16. *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.*

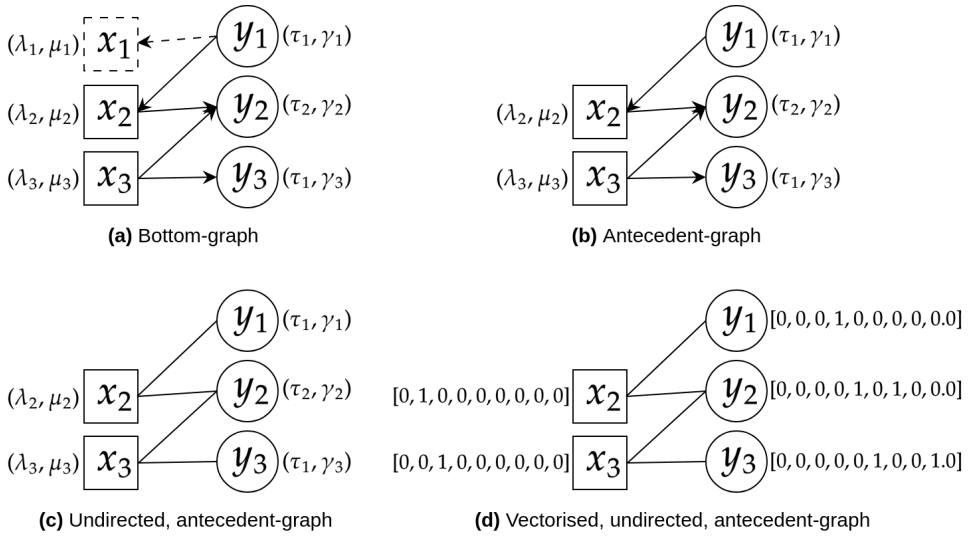
- $X = \{x_1, x_2, x_3\}$, *where* $x_1 = h_x((\lambda_1, \mu_1))$, $x_2 = h_x((\lambda_2, \mu_2))$ *and* $x_3 = h_x((\lambda_3, \mu_3))$
- $Y = \{y_1, y_2, y_3\}$, *where* $y_1 = h_y((1.0, \mathbb{R}))$, $y_2 = h_y((white, \#colour))$, $y_3 = h_y((1.0, \#\mathbb{R}))$
- $E = \{(y_1, x_1), (y_1, x_2),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

and the vertex-labelling ψ *is given 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*):*



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

Definition 5.15 (Graph Transformation)**.** *We define a transformation function over* $\mathcal{G}$ *as* $TransformGraph(G) = Vectorise(UGraph(Antecedent((G, \psi))))$.

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

Definition 5.16 (BotGNN Graphs)**.** *Given a data instance* $e \in \mathcal{E}$*, background knowledge* B*, a set of modes* M*, a depth-limit* d *as before, we define* $BotGNNGraph_{B,\mathrm{M},d}(e) \equiv BotGNNGraph(e) = TransformGraph(BotGraph_{B,\mathrm{M},d}(e))$.

Figure 5.2 summarises the sequence of computations used in this chapter. We will use the term $BotGNN$ to describe GNNs constructed from BotGNN graphs.

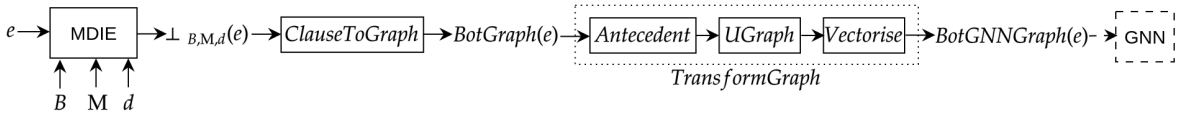
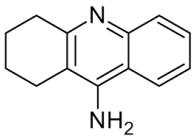


Figure 5.2: Construction and use of bottom-graphs for use by GNNs in this chapter. We note that constituting the transformation of bottom-graphs are for the GNN implementations used in this chapter.

Procedure 6 and Procedure 7 use the definitions, we have introduced, to construct and test $BotGNN$ models. In practice, Step 3 of Procedure 6 and Step 3 of Procedure 7 involve some pre-processing that converts the information in BotGNN graphs into a syntactic form suitable for the implementations used. The procedures assume that data provided as graphs can be represented as definite clauses (Steps 3 in Procedure 6 and 3 in Procedure 7). We illustrate this with an example.

Example 5.17. *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, respectively) 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 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 6 Procedure to construct a *BotGNN* model, given training data $D_{tr} = \{(g_i, y_i)\}_1^N$, where each g_i is a graph and y_i is the class-label for g_i, Background knowledge B, modes M, depth-limit d, and some procedure $TrainGNN$ that trains a graph-based neural network

---

1: **procedure** TrainBotGNN(D_{tr}, B, M, d, $TrainGNN$)
2: $D'_{tr} = \{ (g'_i, y_i) : (g_i, y_i) \in D_{tr}$, e_i be a ground definite-clause representing g_i, $g'_i = BotGNNGraph_{B,\mathrm{M},d}(e_i) \}$
3: Let $BotGNN = TrainGNN(D'_{tr})$
4: **return** $BotGNN$

5.2.5 Note on Differences to Vertex-Enrichment

In this section we clarify some differences of BotGNNs with the approach of vertex-enrichment in GNNs (or VEGNNs) discussed in Chapter 4. An immediate difference is in the nature of the graphs handled by the two approaches. Broadly, VEGNNs require

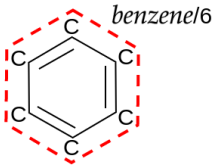
Procedure 7 Procedure to obtain predictions of a *BotGNN* model on a data set given background knowledge B, modes M, depth-limit d, and data D consisting of a set of graphs $\{g_i\}_1^N$

1: **procedure** TESTBOTGNN(D, B, M, d)
2: Let $D' = \{(g_i, g_i') : g_i \in D,\ e_i$ is the definite-clause representation of $g_i,\ g_i' = BotGNNGraph_{B,\mathrm{M},d}(e_i)\}$
3: Let $Pred = \{(g_i, \hat{y}_i) : (g_i, g_i') \in D', \hat{y}_i = BotGNN(g')\}$
4: **return** $Pred$

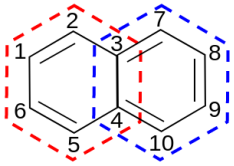
data in a graphical form. VEGNNs retain the most of the original graph-structure, but modify the feature-vectors associated with each vertex of the graph (more on this below). BotGNNs on the other hand do not require data to be a graph. Instead, any data representable as a definite clause are reformulated 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 an example below.

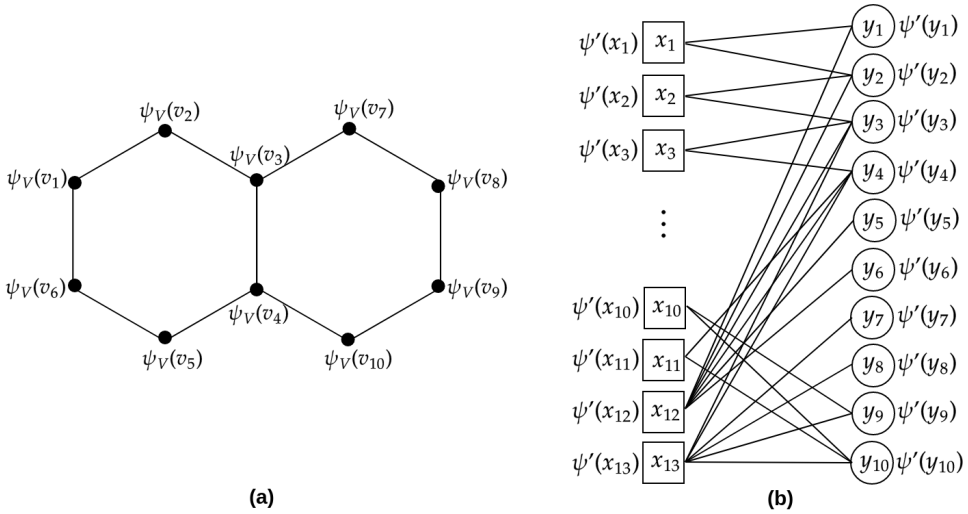
Example 5.18. *Suppose we consider a molecule containing the atoms and bonds, and we want to include the* 6*-ary relation of a benzene ring as shown below.*



In VEGNNs, 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 Chapter 4, 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, 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 in figures (a) and (b) respectively:



$\psi_V/1$ in (a) refers to the vertex-encoding function in Chapter 4, and ψ' in (b) refers to the function defined in Defn. 5.14. For the experimental data in this chapter, the vertex-encoding in Chapter 4 results in vectors whose dimensions are about 10 times more than the $\psi_V/1$ from Defn. 5.14.

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.

5.3 Empirical Evaluation

5.3.1 Aims

Our aim in this section is to investigate the utility of using *BotGNN*s as a technique for including domain-knowledge. That is,

- We investigate whether the performance of a BotGNN that includes domain-knowledge using MDIE is better than the performance of a GNN that does not include domain-knowledge.

As before, later in the chapter, we will also provide additional comparisons against all other methods developed so far in the dissertation.

5.3.2 Materials

Data and Background Knowledge

For the empirical evaluation of our proposed BotGNNs, we use the same 73 benchmark datasets that are used in our studies on DRMs (in Chapter 3) and VEGNNs (in Chapter 4). We use the same background knowledge used in the previous two chapters. Each dataset consists of a set of chemical compounds, which are then converted into bottom-graphs. A simple summary of the resulting bottom-graph datasets is provided below.

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

Figure 5.3: Dataset summary. 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. The last 3 columns are the average number of X, Y and E in each bottom-graph in a dataset.

Algorithms and Machines

The datasets and the background knowledge are written in Prolog. We use the ILP engine, Aleph [Sri01] for the construction of bottom-clauses using MDIE. A Prolog program is then used to extract the relations and ground terms from the bottom-clause. We use YAP compiler for execution of all our Prolog programs. The files containing the relations and ground terms are then parsed by UNIX and MATLAB scripts to construct bottom-graph datasets in a format suitable for GNNs for which we follow the format prescribed

in [KKM+16]. These details are mainly representations of adjacency matrix, vertex labels (feature vectors), 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 [FL19]—a popular geometric deep learning extension for PyTorch [PGM+19] 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.

5.3.3 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 6 (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 $D'_{Te} = \{g_i : (g_i, y_i) \in D_{Te}\}$;

(5) Obtain the predictions for D'_{Te} of $BotGNN$ using Procedure 7 (TestBotGNN) with background knowledge B, modes M, and depth-limit d;

(6) Obtain the predictions for D'_{Te} using GNN; and

(7) Compare the performance of $BotGNN$ and GNN.

We closely follow the method used in Chapter 4 for the construction of BotGNNs. Relevant details are as follows:

- 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.

- The general workflow involved in GNNs was described in Chapter 4 (refer section 4.1). A diagram of the components involved in implementing that workflow for constructing a BotGNN is shown in Figure 5.4. As shown in the figure, a GNN in our implementations 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).

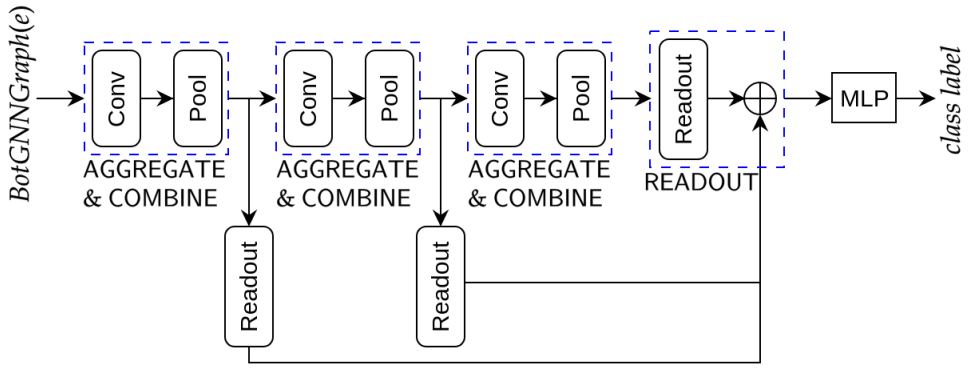


Figure 5.4: Components involved in implementing the workflow in section 4.1 for BotGNN models. 'Conv' and 'Pool' refer to the graph-convolution and graph-pooling operations, respectively. The 'Readout' operation constructs the representation of a graph by accumulating information from all the vertex 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.

- The graph pooling block uses self-attention pooling [LLK19] with a pooling ratio of 0.5. We use the graph-convolution formula proposed in [KW17] 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.* [CVJ+18]. 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 ($2m$) 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 optimiser [KB15] 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 $(\beta_1, \beta_2) = (0.9, 0.999)$.

- The maximum number of training epochs is 1000. The batch size is 128.

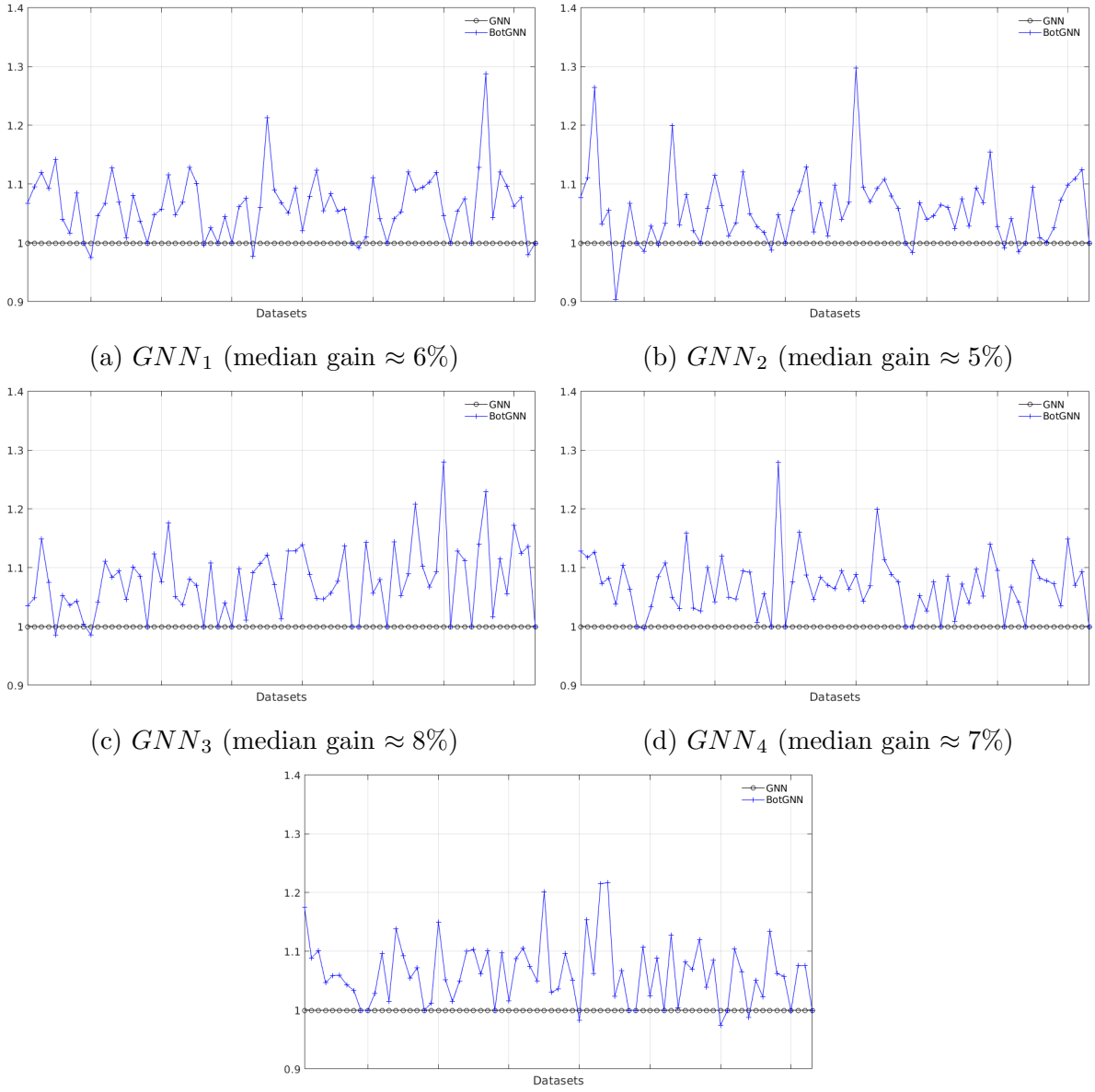
- We use an early-stopping mechanism [Pre98] to avoid overfitting during training. The resulting model is then saved and can be used for evaluation on the independent test-set. The patience period for early-stopping is fixed at 50.

- 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).

5.3.4 Results

The quantitative comparisons of predictive performance of *BotGNN*s against baseline *GNN*s are presented in Figure 5.6. The tabulation shows number of datasets on which *BotGNN* has higher, lower or equal predictive accuracy. The principal conclusion from these tabulations is: *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 Figure 5.5. This is irrespective of the variant of GNN used, suggesting that the technique is able to usefully integrate domain-knowledge into learning. In overall, the results here provide sufficient evidence that incorporating domain-knowledge into deep neural networks significantly improves their predictive performance.



(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 $\approx 6\%$)

Figure 5.5: 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.

Some Additional Comparisons

We now turn to a question of practical interest: Are $BotGNN$s better than $VEGNN$s that are studied in the preceding chapter? To answer this question, we provide some comparisons $BotGNN$s against $VEGNN$s for the same set of datasets and domain-knowledge. The methodology adopted for this comparison is similar to what was done for comparing $BotGNN$s against GNNs (refer subsection 5.3.3). Figure 5.7 provides a tabulation of this comparison for all the variants of GNNs. The results suggest that

GNN Variant	Accuracy ($BotGNN$ vs. GNN) Higher/Lower/Equal (p-value)
1	59/5/9 (< 0.001)
2	59/8/6 (< 0.001)
3	61/2/10 (< 0.001)
4	63/1/9 (< 0.001)
5	60/4/9 (< 0.001)

Figure 5.6: Comparison of predictive performance of $BotGNN$s against GNNs. The tabulations are the number of datasets on which $BotGNN$ has higher, lower or equal predictive accuracy (obtained on a holdout set) than GNN. Statistical significance is computed by the Wilcoxon signed-rank test.

$BotGNN$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.

GNN Variant	Accuracy ($BotGNN$ vs. $VEGNN$) Higher/Lower/Equal (p-value)
1	54/11/8 (< 0.001)
2	61/9/3 (< 0.001)
3	54/10/9 (< 0.001)
4	55/11/7 (< 0.001)
5	52/9/12 (< 0.001)

Figure 5.7: Comparison of predictive performance of $BotGNN$s against $VEGNN$s. The tabulations are the number of datasets on which $BotGNN$ has higher, lower or equal predictive accuracy (obtained on a holdout set) than a $VEGNN$. Statistical significance is computed by the Wilcoxon signed-rank test.

In a previous section (subsection 5.2.5) we have described differences between BotGNNs and VEGNNs arising from an encoding of the data into a bipartite graph representation. Possible reasons for the significant difference in performance of $BotGNN$s and $VEGNN$s 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 subsection 5.2.5. A further difference, not apparent from the tabulations, is in the feature-vectors associated with each vertex. For the data here, vertex-labels for VEGNNs described in Chapter 4 results in each vertex being associated with a 1400-dimensional vector. For BotGNNs, this is about 130.

Next, we turn to the second question: Are *BotGNN*s better than propositionalisation? To answer this question, we provide a comparison of the predictive performance of *BotGNN*s against the two different methods studied in Chapter 3, which are: (a) DRMs constructed with relational features sampled using the hide-and-seek sampling, and (b) MLPs constructed relational features constructed using Bottom-Clause Propositionalisation (BCP [FZG14]). For (a), the quantitative comparison between BotGNNs and DRMs is provided in Figure 5.8 with different number of input features for a DRM. At the outset, it may look like DRMs are as good as BotGNNs, if provided with sufficient number of relational features; and there arises one more question: Why should we bother using BotGNNs at all? BotGNNs are more powerful and capable than DRMs for at least the following reasons: (1) DRMs need to be provided with a sufficient number of relational features (In Figure 5.8 this turns out to be 1000) to match the same level of performance as a BotGNN; (2) DRMs need to be provided with an expressive set of relational features to reach the same level of performance as a BotGNN; (3) For a DRM, there is significant computational effort is required to draw these 1000 features using sampling. As discussed in Chapter 3, the sampling procedure incurs a huge computational cost to select a set of 1000 features where selecting just one relational feature requires the following computational steps: sampling a lot more than one feature, evaluating them for their utilities, and discarding the features with bad utilities. Whereas BotGNNs do not involve any such sampling step and therefore the computational cost remains relatively minimal. So our answer to the primary question of whether BotGNNs do more than propositionalisation is a "yes".

GNN Variant	Accuracy (*BotGNN* vs. *DRM*) Higher/Lower/Equal (p-value)			
	No. of features = 50	$\cdots$	No. of features = 500	No. of features = 1000
GNN_1	64/8/1 (< 0.001)	$\cdots$	46/27/0 (0.15)	39/34/0 (0.98)
GNN_2	63/9/1 (< 0.001)	$\cdots$	31/42/0 (0.17)	29/44/0 (0.05)
GNN_3	65/7/1 (< 0.001)	$\cdots$	42/31/0 (0.66)	37/36/0 (0.46)
GNN_4	65/7/1 (< 0.001)	$\cdots$	43/30/0 (0.18)	40/33/0 (0.72)
GNN_5	67/5/1 (< 0.001)	$\cdots$	44/29/0 (0.26)	36/37/0 (0.83)

Figure 5.8: Quantitative comparison of predictive performance of *BotGNN*s against *DRM*s. *DRM* denotes the Deep Relational Machine constructed using propositionalisation of relational features. The relational features for a DRM are sampled using the hide-and-seek sampling strategy proposed in Chapter 3. The comparative performance of BotGNNs against DRMs starts worsening after 1000 features, which are not shown here. The tabulations are the number of datasets on which *BotGNN* has higher, lower or equal predictive accuracy on a holdout-set. Statistical significance is assessed by the Wilcoxon signed-rank test.

Our answer is further supported by the quantitative comparison provided in Figure 5.9

where the number of relational features constructed using BCP is approximately 18000-52000 across the 73 datasets, which is far more than the length of the graph representation constructed by a GNN. The results here reaffirm that though propositionalisation based techniques are simple, and they require significant computational overhead to perform well in practice.

GNN Variant	Accuracy (*BotGNN* vs. BCP+MLP) Higher/Lower/Equal (p-value)
1	58/10/5 (< 0.001)
2	58/11/4 (< 0.001)
3	61/6/6 (< 0.001)
4	62/6/5 (< 0.001)
5	60/6/7 (< 0.001)

Figure 5.9: Comparison of predictive performance of BotGNNs with an MLP constructed using BCP-based relational features. The tabulations are the number of datasets on which a *BotGNN* has higher, lower or equal predictive accuracy (obtained on a holdout set) than BCP+MLP.

Further, 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 Figure 5.10).

Method	Vector Representation	Vector Dimension (Range)
BotGNN	Real, dense	16–256
DRM	Boolean, sparse	1000s
BCP+MLP	Boolean, very sparse	18000–52000

Figure 5.10: 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). 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.

Finally, we turn to a question that we have so far not considered in the dissertation, namely: how well does a deep neural network with domain-knowledge compare against an ILP learner? ILP represents the pre-eminent approach for dealing both with relational data and symbolic domain-knowledge. Given the comparisons we have shown

so far, we will treat *BotGNN*s as the state-of-the-art for direct inclusion of symbolic domain-knowledge into a GNN. Figure 5.11(a) shows a routine comparison of *BotGNN*s against the Aleph system [Sri01], which is probably the most widely-used ILP engine to date [CD20]. 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 Figure 5.11(b), which compares BotGNN performance on older benchmarks for which ILP results after parameter optimisation are available. These suggest that BotGNN performance to be comparable to an ILP approach with optimised parameter settings.[14]

GNN Variant	Accuracy (*BotGNN* vs. ILP) Higher/Lower/Equal (p-value)
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)

(a)

Dataset	ILP	BotGNN
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

(b)

Figure 5.11: Comparison of predictive performance of BotGNNs with an ILP learner (Aleph system): (a) Without hyperparameter tuning in Aleph; (b) With hyperparameter tuning. 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 [SKB03])

5.4 Summary

In this chapter, we proposed a general technique to construct graph neural networks from relational data and symbolic domain-knowledge. Our experiments here re-validate our claim on importance of the role of domain-knowledge. The significant improvements in performance that we have observed support our claim that when training data available are small, deep neural network models can benefit significantly from the inclusion of

[14] We note that parameter screening and optimisation is not routinely done in ILP. In [SR11] 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."

domain-knowledge. We have also provided additional results that suggest that the technique may be doing more than a simple "propositionalisation" or "vertex-enrichment". The linking of symbolic and neural techniques, as is done in this chapter, provides an interesting direction of research to neural-symbolic modelling.