

# Graph Neural Networks as the Copula Mundi between Logic and Machine Learning: a Roadmap

Andrea Agiollo, Giovanni Ciatto and Andrea Omicini

*Dipartimento di Informatica – Scienza e Ingegneria (DISI), ALMA MATER STUDIORUM—Università di Bologna, Italy*

## Abstract

Combining machine learning (ML) and computational logic (CL) is hard, mostly because of the inherently different ways they use to represent knowledge. In fact, while ML relies on fixed-size numeric representations leveraging on vectors, matrices, or tensors of real numbers, CL relies on logic terms and clauses—which are unlimited in size and structure.

Graph neural networks (GNN) are a novelty in the ML world introduced for dealing with graph-structured data sub-symbolically. In other words, GNN pave the way towards the application of ML to logic clauses and knowledge bases. However, logic knowledge can be encoded into graphs in several ways, and which is the wisest one heavily depends on the particular task at hand.

Accordingly, in this paper, we provide the following contributions: (i) we elicit a number of problems from the field of CL that may benefit from as many graph-related problems where GNN has been proved effective, (ii) we exemplify the application of GNN to logic theories via an end-to-end toy example, to demonstrate the many intricacies hidden behind such practice; (iii) we discuss the possible future direction concerning the application of GNN to CL in general, pointing out opportunities and open issues.

## Keywords

graph neural networks, machine learning, embedding, computational logic,

## 1. Introduction

Artificial Intelligence (AI) has gained significant importance in our ever evolving and technology-focused world. Rising popularity for this field can be attributed to the overwhelming success of sub-symbolic techniques like deep learning. However, along with AI increase in popularity, there exists public concern related to the relevant role that intelligent systems will bear in human society, in particular for the lack of understandability of AI systems.

Whenever intelligent systems play critical roles within human society, they should be made understandable clearly from a human perspective. This need has been taken under deep consideration by the XAI (eXplainable Artificial Intelligence [1]) research community. XAI approaches would seemingly require the integration between successful sub-symbolic techniques and symbolic frameworks in order to reach their main goals [2]. Among symbolic approaches that are being examined nowadays, logic-based techniques possibly represent the most straightforward path towards human understanding. The reason behind that is straightforward: symbols are far

---


WOA 2021: Workshop “From Objects to Agents”, September 1–3, 2021, Bologna, Italy (online)

✉ andrea.agiollo@unibo.it (A. Agiollo); giovanni.ciatto@unibo.it (G. Ciatto); andrea.omicini@unibo.it (A. Omicini)

🆔 0000-0003-0531-1978 (A. Agiollo); 0000-0002-1841-8996 (G. Ciatto); 0000-0002-6655-3869 (A. Omicini)



© 2021 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

 CEUR Workshop Proceedings (CEUR-WS.org)

closer to the way our conscious mind works than the vectors, tensors, and algebraic operations sub-symbolic AI is built upon. Along this line, it is essential for the future of AI to harmonise and integrate symbolic – and, in particular, logic-based – and sub-symbolic AI—and, in particular, neural networks.

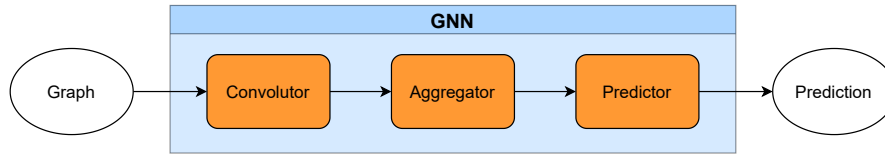
The idea that symbolic and sub-symbolic AI are complementary under a number of dimensions is well understood [3, 4]. However, a general purpose solution to the problem is still lacking. Should we speculate on what is bringing inertia into this field of research, we would argue that the two approaches to AI are fundamentally dual w.r.t. the way they represent information—in short, formulæ vs. tensors. Indeed, a basic requirement of any integrated system involving both symbolic and sub-symbolic processing is the capability to either convert symbols into tensors, or vice versa—possibly both. This problem is hard to formalise in the general case.

In this work we focus on the simpler problem of enabling the sub-symbolic processing of logic knowledge-bases. In particular, we focus on the exploitation of neural networks (NN) as a means to complement computational logic (CL) when it comes to process symbolic data expressed in logic form. NN have indeed proven their strengths in many scenarios, ranging from computer vision to natural language processing, and, more recently, on knowledge graphs and graph-like data.

Accordingly, in this work we study the possible use of graphs as a bridge between the computational logic – among the most prominent branches in symbolic AI – and neural networks—among the most flexible and successful approaches in ML. We consider graphs as the ideal bridge because of their versatility in representing virtually any sort of data structures, there including recursive ones—an aspect which is very common in logics and quite critical in ML.

Accordingly, we investigate the potential and the intricacies of blending CL and graph neural networks (GNN) [5]—a novel category of NN which is particularly well-suited to handle graph-like data. In particular, the aim of the work is to understand how and to what extent CL systems can benefit from GNNs and graph-oriented ML in general. Along this line, the contributions brought by this work are the following:

- we categorise a number of problems from the field of CL which may take great advantage from GNN, given the existence of a clear way to express the same problem in terms of graphs;
- we discuss a number of graph-based tasks which can be suitably tackled via GNN, and discuss how the aforementioned problems can be mapped into such existing tasks;
- we present an end-to-end scenario where GNN are applied to a simple logic knowledge base to detect missing facts via link prediction: the toy example is used to demonstrate the many intricacies hidden behind the application of GNN over logic data;
- finally, we provide a research roadmap that researchers interested in this field may follow for future works, eliciting opportunities which may arise from the integration among CL and GNN.



**Figure 1:** Graph Neural Networks are composed as a cascade of simpler blocks.

## 2. State of the Art

In this section we briefly introduce graph neural networks, along with the tasks these aim at solving and their working principle (Section 2.1). We then give an overall background on computational logic, focussing in particular on the many ways to translate a knowledge base into a graph—in order to make it processable through GNN (Section 2.2).

### 2.1. Graph Neural Networks

In recent years, machine (ML) and deep learning (DL) techniques have disrupted the way complex data-driven tasks – ranging from image classification to speech recognition, natural language processing and many more – are tackled. However, most of these tasks are characterised by a strong reliance on neural networks (NN), which, by construction, can only manipulate data having a *fixed* structure and size—most notably, vectors, matrices, or tensors of real numbers. This may be troublesome in some contexts, given the ever-increasing popularity of applications involving data which cannot be suitably represented by fixed-size, rigid structures such as tensors of numbers. Among the most relevant applications in this category, we can find a number of *graph*-processing scenarios. The reason should be obvious: graphs can hardly be reduced to fixed-sized data structures, unless adjacency matrices are used—which are however quite impractical and often intractable as the dimension of graph increases. To tackle this issue, research effort has focused on extending ML approaches to graph-structured data. Notably, *graph neural networks* (GNNs) are a novel approach to let ordinary NN-based processing be applied to graphs.

GNN are mathematical models accepting as input a graph  $G$  and producing a certain output, which depends on the learning task to be performed. They operate upon *directed* graphs, whose vertices (resp. arcs) are labelled with vectors (or matrices, or tensors) of real numbers, each one carrying further numeric information about the corresponding vertex (resp. arc). Operation may commonly aim to perform any of three wide classes of tasks: *(i)* the classification of similar graphs having different topology, *(ii)* the classification of vertices of unknown graphs, or *(iii)* the identification of missing but statistically probable arcs.

Figure 1 depicts the general architecture of a GNN. It consists of a cascade of three functional blocks (each one composed by one or more layers of neurons) serving specific purposes:

**convolver** – the first block of the GNN, which is in charge of accepting the graph  $G$  as input and producing a new *convoluted* graph  $G'$  as output, having the same topology of  $G$ , where the vector associated with each vertex  $v$  has been replaced by another vector describing the neighbourhood of  $v$ ;

**aggregator** — the convoluted graph  $G'$  is then passed to an aggregator block that produces a fixed-sized representation (e.g. a matrix or a tensor) of the graph  $G'$ —often called *embedding* of  $G$ ;

**predictor** — such embedding produced by the aggregator block, being fixed in size, can be used as the input of an ordinary NN — namely, the predictor block — to solve ordinary ML tasks (e.g. classification or regression) on the original graph  $G$ .

As the main difference among ordinary NN and GNN mostly lies in the convolutor block, here we start our discussion by focussing on the aggregator and predictor blocks, leaving a detailed description of how the convolutor block works to a later paragraph.

The particular shape of the aggregator and predictor blocks heavily depend on the particular learning task that the GNN aims to solve, whereas the convolutor block has a predefined structure, which is essentially the same for all GNN. There are many relevant graph-related learning tasks which can be suitably tackled with GNN, namely:

**graph classification** — given a graph  $G$ , assign a label to the graph—choosing among a predefined set of possible labels. As an example consider a graph representing a chemical molecule where vertices are atoms that compose the molecule and arcs are the chemical bonds between them. In this case, the graph label may predict the molecule hydrophobicity [6], which depends on the atomic properties of the elements that compose the molecule and on the molecule structure.

**node classification** — given a graph  $G$ , assign a label to each vertex of the graph. As an example consider a graph representing published scientific papers, linked if there exists a citation between them, the vertex label may predict the paper subject [7].

**link prediction** — given a graph  $G$ , find missing arcs among couples of vertices. As an example consider a graph representing accounts of a social network, the link prediction may suggest which users of the social network should make a friendship [8].

Accordingly, depending on the particular task at hand, different aggregation functions may be used inside the aggregator block to aggregate the vectors/matrices/tensors associated with each vertex of  $G'$  into a single vector/matrix/tensor. The predictor block is then used to produce the final classification/prediction based on that single vector/matrix/tensor.

As far as *graph classification* is concerned, a possible aggregation function would be the summation among all vertices of  $G'$ . This would provide concise information about the graph as a whole to the *predictor* block, which would then be able to classify it accordingly.

Considering *node classification* instead, a possible aggregation function would be the concatenation among all vertices of  $G'$ . This would provide extensive information about each single vertex of the graph and its neighbourhood to the *predictor* block, which would then be able to classify vertices individually.

Finally, the *convolutor* block is the most interesting component of GNNs and the one we present more in details here. It is here that an operation equivalent to convolution is used to extract knowledge from  $G$  and build a proper embedding of the original graph.

**Graph notation and definitions.** Commonly directed graphs are defined only as a set of vertices  $V$  and a set of arcs  $A$ . However, in practice, both vertices and arcs may be carrier of further information. Consider for instance the graph representation of a chemical molecule: it would be meaningless if the sort of atomic element is not associated with each vertex. The same holds for the details of the chemical bonds among two any atoms of a molecule—which must be associated with the graph’s arcs.

Accordingly, we here consider vertices of a graph to be characterised by specific features called vertex attributes, represented as vectors of the form  $\mathbf{x}_v \in \mathbb{R}^d$ , where  $v$  enumerates the vertices of a graph, and  $d$  is the dimension of all vectors of all vertices. We also assume the existence of total ordering among the vertices in  $V$ , so we may refer to a vertex by its index  $i$ . We denote by  $\mathbf{X} \in \mathbb{R}^{n \times d}$  the matrix of all vertex attributes, attained concatenating each vector  $\mathbf{x}_v$  along a single dimension. There,  $n$  is the number of vertices in  $G$ . We also denote by  $N(v)$  the neighbourhood of a vertex  $v$ , here considered as the set of all vertices  $u$  directly linked to vertex  $v$  by an outgoing arc, i.e.  $N(v) = \{u \in V \mid (v, u) \in E\}$ .

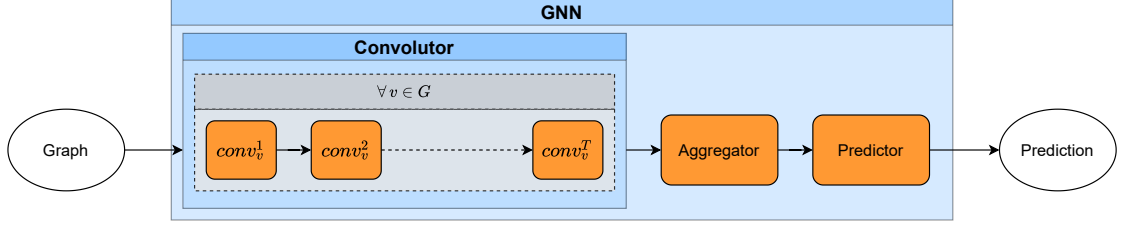
Concerning arcs, we denote by  $a_{v,w} \equiv (v, w) \in A$  the arc connecting vertex  $v$  to vertex  $w$ . Similarly to vertices, we consider arcs in the graph as characterised by specific arc attributes, represented by vectors of the form  $\mathbf{a}_{v,w} \in \mathbb{R}^c$ , where  $c$  is the cardinality of the all vectors of all arcs. Finally, we denote by  $\mathbf{A} \in \mathbb{R}^{m \times c}$  matrix containing all arc attributes. There,  $m$  is the number of arcs in  $G$ .

It is worth to be mentioned that similar definitions of arcs and arc attributes can be found when hyper-graphs – i.e. graphs where arcs may connect more than two vertices – are considered.

**Convolver.** Convolution has been extensively exploited in DL to let NN look at data at different scales and at different levels of granularity—a capability that has shown its success in traditional ML tasks. The underlying idea behind convolutional NN is to let each layer synthesise the information of a *local* area of the previous layer feeding its input. Following this pattern, convolution is applied over each possible portion of the datum received as input. The results into a novel datum, structurally analogous to the one received in input, yet carrying more information. The same procedure may be repeated several times. Successive steps of convolution are then used to build on top of local information, generalising over the importance of local features over the original datum.

The convolver block plays a similar role in GNN. W.r.t. the input graph  $G$ , in particular, the convolver block aims to extract the relevance of each vertex w.r.t. the whole graph  $G$ . The same operation, repeated for each vertex of  $G$ , produces a new graph  $G'$ , such that each vertex carries a lot more information.

While being mathematically well-defined over Euclidean spaces, the application of convolution operation to non-Euclidean data is not straightforward. Therefore, an equivalent notion of convolution is defined over graphs in order to compute the relevance of each vertex w.r.t. to its neighbours. As depicted in Figure 2, the convolver block of a GNN is composed by a cascade of graph convolutional operations. In particular, a graph-convolutional operation is defined over a single vertex  $v$  of a graph  $G$ , and its neighbourhood  $N(v)$ . More precisely, the graph-convolution operation can be described as relying on three successive phases, to be performed for each  $v$ :



**Figure 2:** Graph Neural Networks cascade a series of graph convolution operations to extract relevant information from the graph.

**propagation** – the information  $\mathbf{x}_{v'}$  belonging to each vertex  $v' \in N(v)$  is weighted by the information  $\mathbf{a}_{v,v'}$  belonging to the arc among  $v$  and  $v'$  and then propagated to vertex  $v$ ;

**aggregation** – the information propagated from each vertex  $v' \in N(v)$  to  $v$  is aggregated using a *parametric* aggregation function;

**transformation** – the aggregated information corresponding to vertex  $v$  is transformed into a new embedding vector and assigned back to vertex  $v$ , as its new state  $\mathbf{x}'_v$ .

The single convolution operation is applied in parallel to each vertex in  $G$ . The whole procedure is repeated  $T$  times. The overall effect of step  $t$  is the production of a new graph  $G^t$  having the same vertices and arcs of  $G$ , where the  $i^{th}$  vertex at step  $t + 1$  carries a more convoluted information than the same vertex at step  $t$ .

If the total amount  $T$  of successive graph-convolutions is fixed, the whole cascade of convolutions can be delegated to feed-forward NN. When this is the case, the NN architecture consists of  $T$  layers, where each layer  $t$  of the GNN produces a different graph  $G^t$  as output. The outcome of the convulator block as a whole is therefore  $G^T$

More formally, the relation tying each layer of the NN with its successor is captured by the following recursive equation:

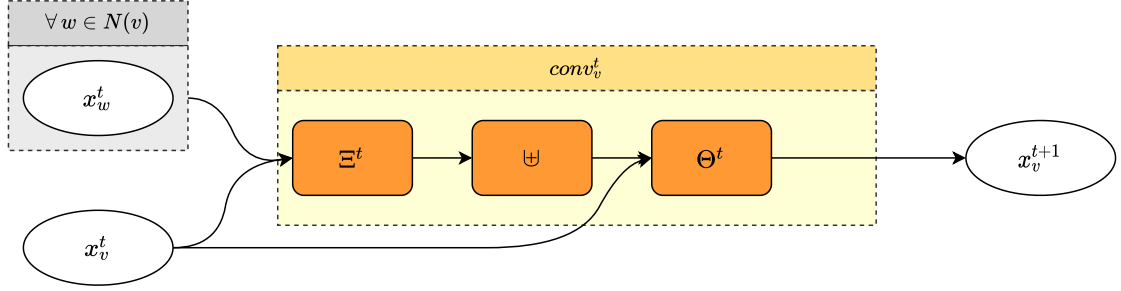
$$\mathbf{x}_v^{t+1} = \Theta^t \left( \mathbf{x}_v^t, \biguplus_{w \in N(v)} \Xi^t(\mathbf{x}_v^t, \mathbf{x}_w^t, \mathbf{a}_{v,w}) \right) \quad (1)$$

where functions  $\Xi^t$ ,  $\biguplus$ , and  $\Theta^t$  represent the *propagation*, *aggregation*, and *transformation* phases respectively.

Function  $\Xi^t$ , in particular, propagates the information belonging to all neighbours  $w \in N(v)$  of vertex  $v$  through the arc that connects the two. This function must be differentiable and parametric – to be amenable of optimisation through the back-propagation algorithm [9] –, other than layer-specific and shared among all vertices of the graph.

Function  $\biguplus$  aims to aggregate the information received by each vertex  $v$  from its neighbourhood. For this reason, it must be variadic and permutation invariant, other than being shared among all layers and all vertices of the graph.

Finally,  $\Theta^t$  is a differentiable, parametric, and layer-specific function, aimed at aggregating neighbourhood information to compute the vertex attributes for vertex  $v$  at step  $t$ . Figure 3 shows the block diagram of the graph convolution operation at step  $t$  for vertex  $v$ .



**Figure 3:** Graph convolution receives in input the information of vertex  $v$  and all its neighbors  $w \in N(v)$  and outputs a new embedding for vertex  $v$ . The convolution at layer  $t$  of the GNN is applied in parallel to all vertices of the graph  $G$ , to obtain a new graph embedding.

The set of vertex features at a specific step  $t$  – i.e.  $\mathbf{x}_v^t$  – is called “graph embedding at step  $t$ ”. The graph embedding at layer 0 – i.e. input of the GNN – is the set of all vertex attributes  $\mathbf{X}^{t=0} = \mathbf{X}$ .

The many graph-convolutional layers composing the convolutor block of a GNN aim to produce a meaningful graph embedding for the learning task at hand—which is then a responsibility of the predictor block. Accordingly, the optimisation of the GNN follows an end-to-end fashion, with *convolutor*, *aggregator* and *predictor* being optimised together. This meaning that differentiable parametric functions inside the GNN – e.g.  $\Xi^t$ ,  $\Theta^t$  – are optimised by minimising a task-dependent loss function  $\mathcal{L}$ .

## 2.2. Logic Theories as Graphs

Computational Logics (CL) essentially deals with logics as a means for computing [10]. Provided that knowledge can be expressed in terms of logic theories (a.k.a. knowledge bases, KB), made up of several logic clauses, CL endows software agents with automated reasoning capabilities, via many sorts of *inference* rules.

Despite the variety of algorithms and procedures provided by CL to attain automated reasoning, a logical formalism aimed at representing knowledge and expressing inference rules is always the underlying requirement. The language of first order logic (FOL) is a common choice to serve this purpose, as well as Horn logic—which is a subset of FOL. They both deal with the syntactic and semantic aspects of automatic knowledge manipulation. On the syntactic level, they dictate how knowledge can be represented as formulæ involving terms, predicates, relations, logic connectors, variables, and quantifiers. On the semantic level, they govern how inferences can be attained over those formulæ, via ad-hoc algorithms.

Our focus here is on the syntactic level, and, in particular, on the many ways to let a (G)NN accept a knowledge base as input—to perform sub-symbolic computations on it. A fundamental aspect along this line regards the way knowledge bases can be encoded into graphs—and, possibly, vice versa. Accordingly, we briefly recall the definitions of *clause* and *knowledge base* from CL, and we discuss the many ways a graph can be attained out of them. In particular, and without lack of generality, we focus on Horn logic to simplify the notation, as well as our discussion.



**Horn clauses** A Horn clause is a formula of the form  $h \leftarrow b_1 \wedge \dots \wedge b_n$  where all  $b_i$  (for  $i = 1, \dots, n$ ) as well as  $h$  are predicates, i.e. statements (or compositions of statements) about one or more entities, represented by logic terms. Predicates may be atomic – e.g.  $p(t_1, t_2, \dots)$  s.t. each  $t_j$  is a term –, or compositions of simpler predicates via logic connectives or other predicate symbols—e.g.  $\forall, \rightarrow, >, fatherOf$ , etc. Terms may be constants (e.g. numbers, or strings), variables (i.e. placeholders for unknown terms), or compositions of simpler terms via functor symbols—e.g.  $f(t_1, t_2, \dots)$  s.t. each  $t_j$  is a term. A term containing no variable – neither directly nor recursively – is called *ground*, similarly to any clause whose predicates only contain ground terms. A knowledge base is a collection of clauses. Similarly to clauses, knowledge bases are ground if they only contain ground clauses.

**KB as graphs.** Clauses can be encoded into graphs in several ways and to serve disparate purposes. Generally speaking, KB can be encoded into graphs by aggregating the graphs attained by encoding all clause therein contained. In all such cases, encoding schemas can act at either the semantic or at the syntactic level. Here, we briefly enumerate possibilities operating on each level.

**Syntactic level.** Encoding schemas in this category capture static relationships inferable from the mere syntax of clauses and KB.

Abstract syntax trees (AST) are the simplest example of graphs which can be attained from KB. They consist of direct acyclic graphs where each vertex has no more than one ingoing arc. There, vertices are of as many sorts as the possible syntactical categories of which may occur in a KB – namely, theories, clauses, predicates, or terms –, whereas arcs simply describe container-contained relations among vertices. Each vertex may then carry further information, such as the predicate symbol of each predicate – which makes sense for predicates' vertices –, or the functor symbol of each term vertex—which only makes sense for terms' vertices. Edges as well may carry further information, such as the position of a contained vertex into its container—which makes sense for clauses', predicates', and terms' vertices.

Dependency graphs are another kind of graph that may be attained from a KB. They consist of directed graphs where each vertex represents a predicate, and each arc represents a logic dependency among two predicates—meaning that the predicate corresponding to the destination vertex must be proven true before the predicate corresponding to the source vertex, in a resolution process.

**Semantic level.** Encoding schemas in this category capture high level relationships that can be inferred from the actual meaning of a logic theory.

Entity-Relationship (ER) graphs are the simplest kind of graph in this category. They aim at expressing via graphs the same information a ground KB expresses via formulæ. They consist of directed graphs where vertices may either represent entities (i.e. terms) or relationships (i.e. predicates) and arcs represent the participation of an entity into a relationship.

Triplet graphs are another simple way of representing ground theories where all terms are constants and all predicates are either unary or binary. When this is the case, each constant is considered an entity, binary predicates are considered as relations among two entities, and



unary predicates are considered as properties an entity may or may not have. Thus, a graph can be attained by defining a vertex for each different constant in a KB, and arc for each couple of constants involved in at least a binary predicate. Vertices may then carry further information describing which properties (i.e. unary predicates) are true for the corresponding entity (i.e. constant) in the KB, whereas arcs may carry further information describing which relations (i.e. binary predicates) the two entities (i.e. constants) are involved into.

### 3. Processing Logic Knowledge via GNN

In this section we present a research roadmap eliciting the potential bridges among CL and GNN. We firstly identify four relevant tasks from CL where, we believe, sub-symbolic processing may have a role to play. We then discuss how all such tasks can be mapped into as many well-known graph manipulation tasks, for which GNN have already been exploited. Finally, we present a general workflow to be followed whenever logic information must be sub-symbolically processed via GNN, and we elicit the many constraints a designer should keep into account when doing so.

#### 3.1. Logical Tasks

Manipulation of logic knowledge enables the resolution of complex queries via logical inference. There exist, however, relevant tasks which are hard to formalise or solve into the logic realm, because of their numerical nature or algorithmic infeasibility. Here, in particular, we identify four relevant operations on knowledge bases for which, we argue, it is worth investigating sub-symbolic solutions.

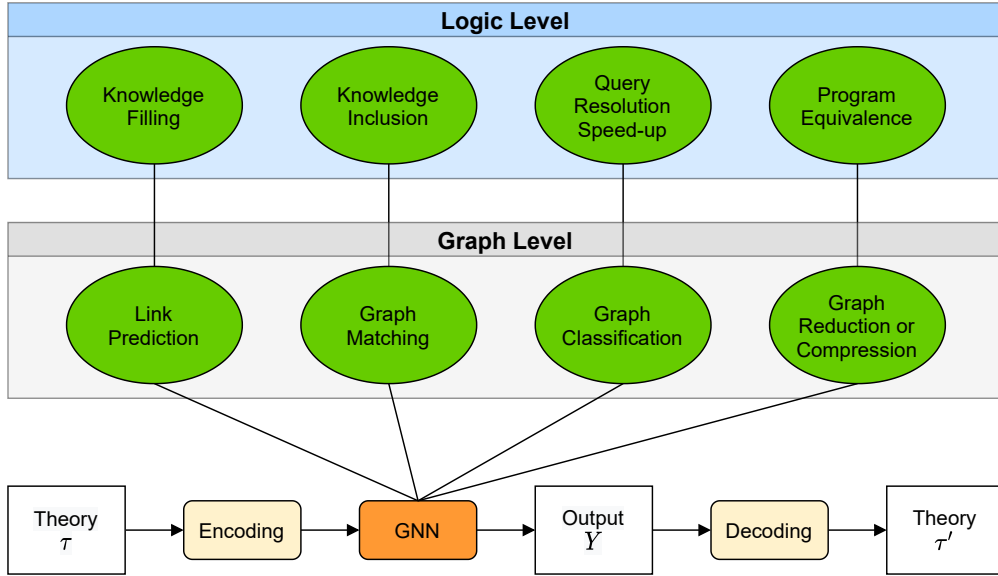
The tasks we consider are presented in the upper box of Figure 4. These are *(i)* knowledge filling, *(ii)* knowledge inclusion, *(iii)* program equivalence, and *(iv)* resolution speed-up.

**Knowledge Filling.** Entities and relations available in a logic theory may sometime lack some instance—e.g. because the human operator handcrafting the theory was imprecise. When this is the case, we consider the knowledge base as *fragmented*.

To deal with such fragmented theories, it may be useful to identify missing relations between existing entities. This task is may be tackled via statistical analysis of the theory under examination, which may lead to the identification of latent relations among entities.

A knowledge filling would be hard to handle symbolically, as logic reasoners commonly struggle in processing knowledge they do not have. While most solvers operate under a closed world assumption – letting them considers as false everything they do not explicitly know to be true –, even the ones operating under an open world assumption do not commonly include mechanisms to generate new knowledge out of thin air. In all such cases, theory the coherence and completeness of the knowledge base is usually considered as an a-priori requirement for logic computations to work properly.

Conversely, in the sub-symbolic realm, the semantic similarities among the many entities and relations involved in a logic theory may be better captured, and this may help reconstructing the missing facts.



**Figure 4:** Proposed roadmap along with its theoretical framework. Logic level and graph level can be mapped directly, in order for logical problems to benefit from sub-symbolical techniques – e.g. GNN – available at the graph level.

Consider for instance the case of a simple theory representing kinship relationships. The lack of a single relation – say that “John and Mary are siblings” – may significantly hinder a solver’s ability to correctly deduce all kinships among entities of a family—e.g. “the sons of John and Mary are cousins”. The solution for this task is not straightforward, thus attracting our attention.

**Knowledge Inclusion.** Knowledge inclusion represents the task checking whether a given theory (usually smaller) is complementary w.r.t. another given theory (usually larger) or not. The same clauses could occur with slightly different shapes—e.g. using different predicates/functor names or different positions arguments in the same predicates.

This task requires the ability to express equivalence or similarity among *groups* of clauses, which is not straightforward [11, 12]. Computing exact solutions to this problem may soon become infeasible as the dimensions of the involved theories increases. Conversely, in the sub-symbolic realm, the same problem may be modelled as a pattern matching problem. This may pave the way towards the computation of *approximate* solutions to the knowledge inclusion problem in reasonable time.

As an example, consider two theories  $\tau_1$  and  $\tau_2$ , both representing family trees. Suppose that  $\tau_1$  express 1<sup>st</sup> degree relatives only, while  $\tau_2$  includes also 2<sup>nd</sup> degree relatives.  $\tau_1$  may consider more/less/different family members w.r.t.  $\tau_2$ , and kinships may also be defined in different ways between the two theories. However,  $\tau_1$  is – logically speaking – a subset of  $\tau_2$ , and we need to detect this property.

**Program Equivalence.** Program equivalence represents the task of computing a simpler and equivalent theory  $\tau'$  starting from a theory  $\tau$ . This may imply removing redundancies and simplifying clauses.

As for the knowledge inclusion task, program equivalence requires a procedure to compare sets of clauses, other than the capability of generating reduced equivalent variants of clauses. It is our opinion that both these procedures may be better expressed into sub-symbolical realm.

In the example of kinship relationships, program equivalence requires to produce a new theory, having fewer rules, yet spanning the same family tree of the original theory. Such a requirement is complex to satisfy and would probably require notions of semantically equivalent sets of kinships—e.g. the set of relations containing only *parent* spans the same family tree of the set of relations  $\{mother, father\}$ .

**Query Resolution Speed-Up.** Logic theories are commonly exploited by logic solvers to draw inferences, via some resolution procedure. The execution time of any query resolution vastly depends on the complexity of the algorithm(s) expressed by the logic theory. To this regard, a number of efficiency tweaks may affect the execution time in the average case. For instance, the solutions to most frequent queries may be cached, or smart strategies may be employed to affect the way the solver explores a solution space. However, caching costs space, while any *rigid* resolution strategy may result efficient on some sorts of queries, while still being slow on some others.

In all those cases, sub-symbolic sub-systems capable to learn from experience can bring about huge benefits. There, a sub-symbolic helper may be trained to predict the outcomes of most frequent queries, thus speeding up queries with constant space requirements. Furthermore, an online learning procedure may be injected into the solver, making it adapt the resolution strategy to the query at hand, on the basis of the experience accumulated via previous queries.

As an example, consider now the simple theory representing kinship relationships. When the number of family members is huge and relations between family members are complex – e.g. fourth grade cousins –, query resolution may suffer from delays. Therefore, it may be interesting to use techniques that aim at speeding up the resolution of queries over such huge theories. Sub-symbolical approaches may ease this task, by compressing theory knowledge to simple and easy-to-handle embeddings.

### 3.2. Graphs as Bridges

Here we discuss the role of GNN in addressing the relevant logic tasks from Section 3.1. In particular, we show how all such tasks can be mapped onto known graph-related problems which can be addressed via GNN. In other words, we comment the upper part of Figure 4.

**Knowledge filling  $\rightarrow$  Link prediction.** The knowledge filling task usually requires semantic knowledge to be taken into consideration. Therefore, to map the knowledge filling task to an equivalent problem over graphs we should consider preserving the semantic information of the theory. We can then assume to map entities of a theory to vertices of a graph. Rules and relations can then be represented as vectorised arcs existing between the graph vertices. Each position of an arc vector represents a specific relation, preserving the original semantic of the

theory. In this scenario, the task of predicting possible missing relations or rules is mapped to the problem of identifying which arcs are missing from the graph. In the graph ML community, this problem is known as link prediction—i.e. given a graph, predict which arcs should or will form between vertices which are not linked yet. Link prediction for knowledge filling is more complex than common link prediction. Indeed, it requires to predict not only arc existence but also the class of link that should form—i.e. which class of relation involves the entities considered.

**Knowledge Inclusion → Graph matching.** As well as for the knowledge filling task, knowledge inclusion requires semantic knowledge of the theory to be taken into account. Therefore, we require the mapping between logic and graphs to preserve the theory semantic. Moreover, knowledge inclusion requires a comparison between two or more theories: entities and relations from a theory should be compared to their counterparts of the other theory and matched upon need.

As done for knowledge filling, let us assume entities to be represented as vertices, and rules or relations as vectorised arcs. The mapping produces as many graphs as the theories available for the inclusion task. Therefore, from a graph perspective, knowledge inclusion is mapped to a graph matching problem. Indeed, the two or more graphs corresponding to their theories counterpart should be matched for some portion of them.

The matching between graphs is still an open research problem, as it is computationally very expensive, but is easier to tackle than rules and entities matching. This is particularly true in situations where entities do not match exactly or rules share analogous semantics but are defined in different forms—e.g. parent and mother.

**Program equivalence → Graph compression.** Given a specific theory, program equivalence aims at obtaining a simpler – smaller – theory that preserve the same expressiveness. Depending on the considered approach the mapping between logic and graph level may bear different requirements. In its simplest form program equivalence requires to simply remove unnecessary relations and rules of a theory to compress it. This approach does not require explicitly the semantic level to be considered while processing the theory. More interestingly program equivalence may also require to map set of rules and relations to a single (or a smaller set of) rules(s). This increased complexity introduces the need for semantic to be taken into account and to be preserved in the mapping from logic to graphs. If we consider the same mapping of previous examples, program equivalence can be linked to graph compression problem. Indeed, obtaining a smaller set of equivalent rules and entities can be done removing or merging together arcs and vertices of the graph theory counterpart.

**Query resolution speed-up → Graph classification.** Query resolution speed-up aims at obtaining faster execution of given queries over a logic theory. It may be helpful for query resolution to maintain the semantic information embedded in the theory. Therefore, the mapping between logic and graphs may benefit from the preservation of semantic information, and generally speaking vastly depends on the requirements of the desired speed-up. Differently from previous tasks, for query resolution speed-up we consider obtaining graphs for queries to

be solved—rather than a single graph for the whole theory. The graph representing a query is matched with the query resolution, considered as the graph label. Following this mapping, the query resolution speed-up is mapped to a graph classification problem, where the label of a graph should be predicted. Any approach can then be leveraged to classify graphs—i.e. obtain query solutions. This approach may not be significant for simple queries applied to small knowledge basis. However, it may result in great speed-up when complex knowledge bases and queries are considered. Indeed, GNNs scalability over large graphs is mostly not an issue, resulting in quick graph classification.

### 3.3. The Workflow Perspective

Here we summarise the general workflow to process logic theories sub-symbolically, via GNN. In a nutshell, the whole workflow consists in *transforming* the problem into the graph domain and let a GNN to do the job, then possibly *transform* back the problem into the logic domain. The same workflow is depicted in the lower part of Figure 4.

Let us assume the overall goal of the whole processing, at a logic level, is to perform a task  $T$ —say, knowledge filling or inclusion. Let us also assume that an adequate mapping exists for  $T$  towards the graph level, such that  $T'$  is graph-related task corresponding to  $T$ , at the graph level. Under such assumptions, the workflow involves the following steps:

1. a logic theory must be firstly encoded into a graph using some suitable graph *encoding schema*;
2. a GNN must be designed and trained to perform task  $T'$ 
  - this step involves choosing the functions  $\Xi^t$ ,  $\uplus$ , and  $\Theta^t$  for the *convolutor* block of the GNN,
  - as well as choosing the particular structures of the *aggregator* and *predictor* blocks of the GNN, according to  $T'$ ;
3. optionally, the output of the GNN shall be decoded back into a logic theory using a suitable graph *decoding schema*;

The emphasised words above represent choice points for the designer. While the possibilities are manifold, it is worth pointing out that each choice affects the others. For instance, while the encoding schema should be chosen by taking the nature of the logic clauses into account, the architecture of the predictor block, as well the choice of functions  $\uplus$  and  $\Theta^t$ , should be tailored on the task  $T'$ —and in particular on its nature under the learning perspective (e.g. whether  $T'$  is classification, regression, or clustering task).

Whether it is needed to perform step 3 (decoding) or not, is another source of constraints. There may be tasks – such knowledge inclusion, corresponding to graph matching – for which the outcome of  $T'$  is a Boolean datum, which needs not a decoding step. Conversely, other tasks may require the outcome of  $T'$  to be transformed back into the logic domain—cf. knowledge inclusion via link prediction. When this is the case, it is of paramount importance to choose an encoding schema which is *invertible*. This implies the encoding and decoding schemas are deeply entangled in the general case.

Summarising, symbolic processing may greatly benefit from the exploitation of sub-symbolic, GNN-based approaches. However, when this is the case, the overall data-processing workflow must be carefully designed, as it involved may inter-dependent design choices.

## 4. Case Study

In this section we describe a case study which puts the theoretical framework introduced in Section 3 to test. We consider the knowledge filling task as the subject of this case study, as we believe it to be a nice introductory example to the world of logic manipulation using GNNs. We proceed to set up our study case over a controlled environment, considering the knowledge basis representing kinship relations—i.e. family tree. We “mutilate” the knowledge base – meaning that we throw away a random part of the knowledge base – and use the remaining part to reconstruct the information removed by using the proposed framework.

### 4.1. Logic to Graph

As already mentioned, measuring the effectiveness of our framework requires to “mutilate” an otherwise exhaustive knowledge base. Theory mutilation can be then attained both at a logical level and at the graph level. In our experiments we mutilate the theory at graph level, to avoid multiple translations between the two levels. We now introduce the mapping function between logic level and graph level used in our experiment.

**Translation to Graph** Let  $\mathcal{C}$  be the set of all ground Horn clauses of the form  $h \leftarrow b_1 \wedge \dots \wedge b_m$  s.t. all  $b_i$  as well as  $h$  are predicates of arity non-greater than 2, and all arguments of all predicates are constant in  $\mathcal{H}$ . We consider  $\tau \in \mathcal{C}^*$  to be a ground theory containing  $N$  clauses and representing a family tree. We then define the *properties* of the theory  $\tau$  to be the set of all the unary predicates mentioned in all clauses of  $\tau$ . The set of properties is considered to be ordered through an index  $k$ , allowing to obtain a property calling it with the corresponding index. We then define the *relations* of the theory  $\tau$  to be the set of all the binary predicates mentioned in all clauses of  $\tau$ . The set of relations is also considered to be ordered through an index  $l$ , allowing to obtain a relation calling it with the corresponding index.

To map the theory  $\tau$  to its graph counterpart  $G_{fill}$ , we first consider all the entities mentioned in all clauses of  $\tau$  and associate a vertex to each entity. Therefore, obtaining a graph  $G_{fill}$  with  $n$  vertices. Vertex features are then built as vectors  $\mathbf{x}_v \in \mathbb{R}^d$ , where  $d$  represents the size of the set of *properties* of  $\tau$ . Vector  $\mathbf{x}_v$  has value at position  $k \in \{1, \dots, d\}$  equal to 1 if property  $k$  holds true for entity  $v$  and 0 otherwise. The obtained vertex feature vector is thus a one-hot encoded vector representing the properties that characterise the entity. Similarly to vertices, arc features are built as vectors  $\mathbf{a}_{v,w} \in \mathbb{R}^c$ , where  $c$  represents the size of the set of *relations* of  $\tau$ . Vector  $\mathbf{a}_{v,w}$  has value at position  $l \in \{1, \dots, c\}$  equal to 1 if relation  $l$  between entities  $v$  and  $w$  holds true and 0 otherwise. The obtained arc feature vector is thus a one-hot encoded vector representing the relations satisfied for couples of vertices. Figure 5 exemplifies the mapping described above for a small family tree. For the experiment we present, the predicates are the same of the figure, along with some added unary predicates – e.g. *has\_siblings*, *is\_parent*, *is\_grandparent*, etc. –, which are used to give more information concerning single entities.

```

sibling(X, Y) :- parent(Z, X), parent(Z, Y), X \= Y.
united(X, Y) :- parent(X, Z), parent(Y, Z), X \= Y.
grandparent(C, D) :- parent(C, E), parent(E, D).
aunt(X, Y) :- female(X), sibling(X, Z), parent(Z, Y).
uncle(X, Y) :- male(X), sibling(X, Z), parent(Z, Y).

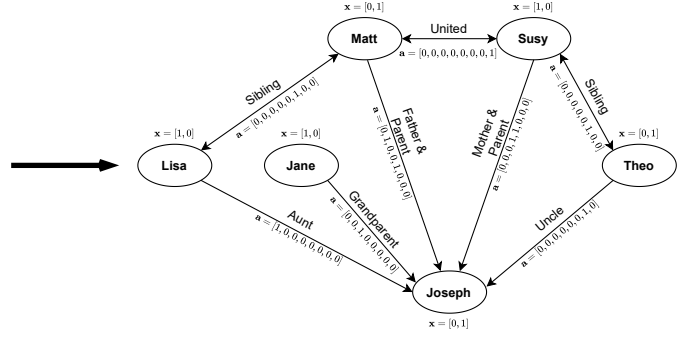
male(matt).
male(theo).
male(joseph).

female(susy).
female(lisa).
female(jane).

parent(matt, joseph).
parent(susy, joseph).
parent(jane, matt).

sibling(susy, theo).
sibling(theo, susy).
sibling(matt, lisa).
sibling(lisa, matt).

```



**Figure 5:** Example of the mapping function used in this case study. vertex attributes ( $\mathbf{x}_v$ ) represent unary predicates – i.e. properties – while arc attributes ( $\mathbf{a}_{v,w}$ ) represent binary predicates—i.e. relations.

Once  $G_{fill}$  is obtained, we proceed to mutilate the theory. Mutilation is attained removing some arcs – i.e. relations – between vertices of the graph. We call the graph obtained through this procedure  $G$  and the set of removed arcs  $A_{test}$ .

The proposed mapping allows constructing uniquely a graph from a grounded knowledge bases and is evidently bijective, as it is possible to reconstruct entities, properties and relations from  $\mathbf{x}_v$  and  $\mathbf{a}_{v,w}$ . However, it still presents some issues, as it requires groundisation of the knowledge bases and it can handle at most binary facts. The former can be considered a mild requirement as it is commonly considered for manipulation of symbolic knowledge. The latter instead has to be attributed to the nature of state-of-the-art GNNs. Predicates of arity greater than 2 would require arcs linking more than two vertices at the time. Graphs having such links are called hypergraphs. These peculiar graphs are still an exception in the world of graph manipulation. There have been proposed very few solutions to handle these graphs [13], presenting strong drawbacks like the absence of arc features.

## 4.2. Graph Manipulation

Given the mutilated theory represented by the graph  $G$ , the task is to train a GNN model capable of mining the missing arcs. The GNN model is required to identify not only the existence of a missing arc between two vertices, but also to classify the arc into its class—i.e. which relation(s) the arc is representing. Given the structural nature of graphs, link prediction can be tackled either considering  $G$  as a unique entity – i.e. *plain approach* – or as a pool of subgraphs—i.e. *subsampling approach*.

**Plain approach.** We consider the graph  $G$  as a whole and predict one solution for each couple of vertices. Indeed, each vertex can be involved in a relation with any other vertex of the graph and the model should be capable of identifying them. For this approach we consider the set of arcs belonging to  $G$  as positive examples, called  $A$ . We then sample a set of negative examples  $\bar{A}$  as all the arcs that are not in  $A$ , nor in  $A_{test}$ . Given in input the graph  $G$ , a GNN model is trained over  $A$  and  $\bar{A}$  to output two predictions:

- A binary matrix  $Y_e \in \mathbb{R}^{n \times n}$ . Where  $n$  is the number of entities in the graph. The value



for position  $\{v, w\}$  is 1 if the GNN predicts that an arc should exist between vertex  $v$  and vertex  $w$  and 0 otherwise.

- A binary tensor  $Y_t \in \mathbb{R}^{n \times n \times d}$ . Where  $n$  is the number of entities in the graph and  $d$  is the number of available facts (kinship relations). The  $d$ -dimensional vector at position  $\{v, w\}$  corresponds to the one-hot encode of the arc type that the GNN predicts between vertices  $v$  and  $w$ .

The GNN model is composed of two graph convolutional layers that extract relevant information from  $G$  and produce a graph embedding  $G'$ . Given the need to predict a solution for each couple of vertices, we define as aggregation function the concatenation of vertices in the graph. The function is iterated over each couple of vertices  $v, w$  producing a vector  $\bar{x}_{v,w} = \mathbf{x}'_v \parallel \mathbf{x}'_w$  that represents the embedding for a possible arc between vertices  $v$  and  $w$ . Two parallel fully connected layers are then used as predictors to predict the existence – i.e.  $Y_e$  – of arc  $v, w$  and its type – i.e.  $Y_t$  – from  $\bar{x}_{v,w}$ . During training, cross-entropy loss  $\mathcal{L}_e$  is computed using  $Y_e$ , while binary cross-entropy over class types is  $\mathcal{L}_t$  is computed using  $Y_t$  [14] The overall loss is then obtained through weighted summation of the two and used to optimize the GNN parameters.

$$\mathcal{L} = \mathcal{L}_e + \gamma \mathcal{L}_t \quad (2)$$

where  $\gamma$  is the balancing factor between the two losses, having the role to balance the importance of predicting arc existence or its type.

**Subsampling approach.** It is also possible to consider the graph  $G$  as a combination of subgraphs each of which is used to predict if one arc exists. For this approach, one subgraph  $G_{sub}$  is obtained for each arc in  $A$ . For each arc  $a_{v,w}$ , the subgraph is obtained by keeping vertices  $v$  and  $w$ , as well as their neighbours  $N(v)$  and  $N(w)$ . The same sampling procedure is repeated for a set of negative examples  $\bar{A}$ . Therefore, following this approach we obtain a set  $\mathcal{G}$  of  $\nu$  graphs, each focused on an arc  $a_{v,w}$ . This approach is similar to the one proposed in [15], which shows promising results on simple link prediction tasks.

A GNN model is then trained over  $A$  and  $\bar{A}$ , receiving in input a graph  $G_{sub}$  at the time, to output two predictions:

- A binary value  $Y_e \in \{0, 1\}$ . The value is 1 if the GNN predicts that the arc  $a_{v,w}$  should exist and 0 otherwise.
- A binary vector  $Y_t \in \mathcal{R}^d$ . Where  $d$  is the number of available facts (kinship relations). The vector corresponds to the one-hot encode of the arc type that the GNN predicts for arc  $a_{v,w}$ .

The GNN model is similar to the one of plain approach. It is composed of two graph convolutional layers that extract relevant information from  $G_{sub}$  and produce a graph embedding  $G'_{sub}$ . Given the need to predict a single solution for each graph, we define as aggregation function the global averaging pooling of vertices in the graph. The global averaging pooling function receives in input the graph  $G'_{sub}$ , producing in output a  $k$ -dimensional vector  $\bar{x}$  where  $k$  represents the dimension of vertex features of  $G'_{sub}$ . Global average pooling is obtained averaging vertex

features of all vertices in the graph  $\bar{\mathbf{x}} = \frac{1}{N} \sum_{v=1}^N \mathbf{x}'_v$ . Two parallel fully connected layers are then used as predictors to predict the existence – i.e.  $Y_e$  – of arc  $v, w$  and its type – i.e.  $Y_t$  – from  $\bar{\mathbf{x}}$ . GNNs of both approaches are built using *PyTorch Geometric* [16] and *Deep Graph Library* [17]. Finally, loss computation remains the same of plain approach.

**Considerations.** For both of the considered approaches it is important to notice that link prediction at graph level is simpler than the initial logical task. Meanwhile, it must be stressed that link prediction over graph is usually considered to be a binary prediction problem only. Indeed, state-of-the-art approaches focus only on the output  $Y_e$ —i.e. if an arc should exist between two vertices. This is due to the nature of common link prediction applications – e.g. chemistry, social networking – where arcs between vertices belong mainly to one category only.

As a consequence, the link prediction problem we face is the mixture of binary classification and multi-label classification. Indeed, there may exist multiple relations linking two entities, which translates to arcs having multiple labels. Multi-label classification problems are not straightforward to tackle due to class overlapping and scalability issues [18].

It would be desirable to have relations semantically distant from each other to aid GNNs in the multi-label classification task. Indeed, classes characterised by similar semantics are less separable and are probably subject to higher misclassification. Moreover, the scalability issue of multi-label classification hinders the performance of GNNs when considering a high number of relation types. Although they could be overcome, these issues must be taken into account during the evaluation of the proposed experiment.

### 4.3. Results

During GNN training procedure we split either  $G$  (plain approach) or  $\mathcal{G} = \{G_{sub}^1, \dots, G_{sub}^\nu\}$  (subsampling approach) between training set and validation set. The former is used for back-propagation, while the latter is used to check GNN performance and save the best model. The best model obtained from training is then applied over  $A_{test}$  to test the final performance of the GNN.

We measure model performance over both predictions tasks—i.e. over  $Y_e$  and  $Y_t$ . We measure how the model behaves for the arc existence prediction task using the Area Under the Curve (AUC), Average Precision (AP) and accuracy metrics. AUC measures the area under the Receiver Operating Characteristic (ROC) curve [19], and can be interpreted as the probability that a randomly chosen missing arc is given a higher score than a randomly chosen pair of unconnected vertices [20].

Evaluation of multi-label classification is not straightforward, as it introduces the notion of partially correct prediction—i.e. those predictions where a subset of the labels are identified correctly, but not all of them. Therefore, to measure how the model behaves in the arc type prediction task we leverage the well known  $F_1$  score, the Exact Match Ratio (EMR), and the Per Example Accuracy (PEA). EMR ignores partially correct predictions, considering them as incorrect and computes the score as the ratio between correct predictions and total predictions. Mathematically,  $EMR = \frac{1}{m} \sum_{i=1}^m I(y_i = z_i)$ , where  $I$  is the indicator function, while  $y_i$  and  $z_i$  are the prediction and label of the  $i^{th}$  arc of  $A_{test}$ . On the other hand, PEA computes the

| Approach    | Edge Existence |       |          | Edge Class |                |       |
|-------------|----------------|-------|----------|------------|----------------|-------|
|             | AUC            | AP    | Accuracy | EMR        | F <sub>1</sub> | PEA   |
| Plain       | 0.786          | 0.786 | 0.786    | 0.143      | 0.148          | 0.768 |
| Subsampling | 0.908          | 0.892 | 0.906    | 0.691      | 0.386          | 0.911 |

**Table 1**  
Performance of the two different approaches over  $A_{test}$ .

accuracy of the single sample  $i$  for each  $i$  in  $A_{test}$  and then average them together to obtain an overall accuracy metric. Mathematically,  $PEA = \frac{1}{m} \sum_{i=1}^m Acc(y_i, z_i)$ , where  $Acc$  is the function computing accuracy, while  $y_i$  and  $z_i$  are the prediction and label of the  $i^{th}$  arc of  $A_{test}$ .

Table 1 shows the performance of the two approaches on the link prediction task over  $A_{test}$ .

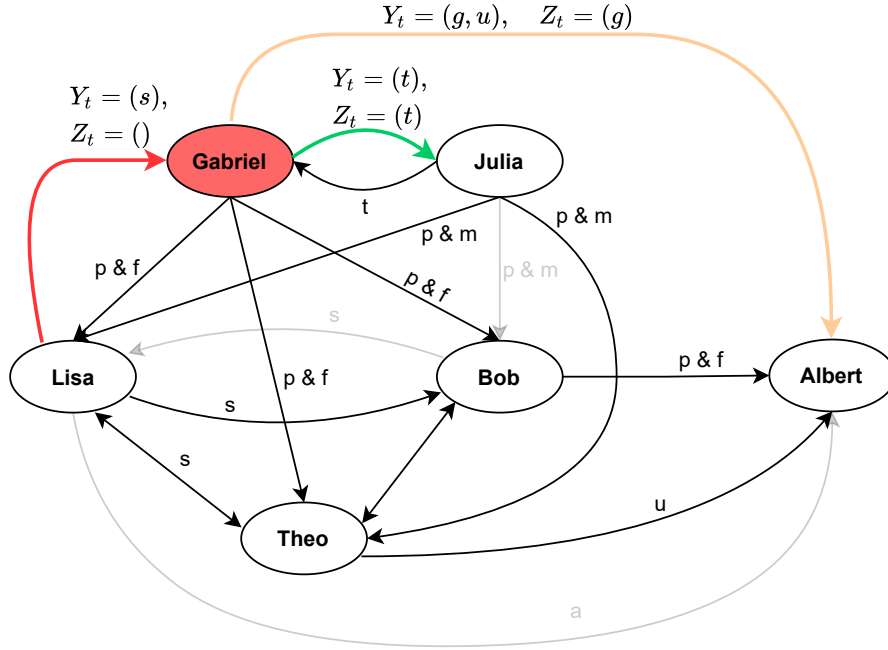
The results obtained demonstrate the effectiveness of the proposed approach. The prediction of arc existence is successful with acceptable level of performance for the plain approach, while being highly successful when graph subsampling is applied. On the other hand, for the arc type prediction task we can notice the clear superiority of the subsampling approach. Indeed, the subsampling approach gives higher metrics for all the scores considered.

Subsampling approach superiority can probably be attributed to the setup of the learning task. For the plain approach, the same input  $G$  is used to predict arcs belonging to training, validation and test sets, increasing the tendency of model overfitting over train set. On the other hand, subsampling approach allows to assign a different input  $G \in \mathcal{G}$  to each arc to predict. These inputs are different between training, validation and test sets, therefore allowing the model to train more easily, avoiding possible overfitting issues.

Given the number of possible arcs types (8) and their possible overlapping – e.g. *parent* and *father* –, the performance obtained by the subsampling approach are very satisfactory. The proposed model is capable to exactly match an arc to its label – i.e. exact match over the 8 types that define the arc – nearly 70% of the times, while the single arc types are predicted correctly more than 90% of the times. These results show the effectiveness of both the GNN model and the proposed theoretical framework of Section 3.

Figure 6 shows an example of three arc predictions involving a single entity. We picked *Gabriel* as an entity since it allows to show the possible three outcomes of a prediction. An arc prediction may be completely correct, as it is for *Gabriel*  $\rightarrow$  *Julia*. There are also partially correct predictions, where the correct labels are a subset of the predicted ones or vice versa. *Gabriel*  $\rightarrow$  *Albert* shows a partially correct prediction. Finally, there may be arcs that are not correctly predicted, as it is for *Lisa*  $\rightarrow$  *Gabriel*.

Figure 6 allows also to highlight a peculiar property of the model. The *Lisa*  $\rightarrow$  *Gabriel* prediction considered as wrong is actually a *daughter* relation. This prediction is considered wrong since nor *daughter* nor *son* are defined in the original theory  $\tau$ . Indeed, to solve the knowledge filling task we rely only on the predicates already defined, not being able to generate new predicates. However, the model being able of predicting this relation is symptom of the GNN ability to understand the semantic of  $\tau$ . This GNN capability is encouraging, as it may help in tasks such as the discovery of new predicates.



**Figure 6:** An example of three arc predictions involving entity *Gabriel*. Vertices represent entities and arcs represent relations, which can be aunt (a), father (f), grandparent (g), mother (m), parent (p), sibling (s), together (t), uncle (u). Black arcs and relations identify arcs in  $G$ , while gray arcs are arcs belonging to  $A_{test}$  and used for testing. Coloured arcs are the ones belonging to  $A_{test}$  and involving *Gabriel*. For these arcs  $Y_t$  and  $Z_t$  represent model prediction(s) and label(s) respectively. Green colour means the arc is predicted correctly, orange identifies partially correct arcs, while red pinpoint arcs wrongly predicted.

**Role of overlapping classes.** As it can be seen in Figure 5, the results obtained for Table 1 are influenced by class overlapping. Indeed, some considered arc types are semantically similar—e.g. *mother* and *parent*. Therefore, their vectorial representation produce label overlapping, which may affect GNN performance. Indeed, ML models suffer hardly-separable data.

To study the effect of semantically similar classes on our approach, we consider a new theory  $\tau'$  identical to  $\tau$  – i.e. representing the same family tree –, but formulated without semantically similar predicates. Practically speaking this requires to remove the notion of *father* and *mother* from  $\tau$  and redefine kinships using only the notion of *parent*.

We apply the same approaches of Section 4.2 to the new theory  $\tau'$  and measure their performance. Removing class Table 2 shows the performance of the two approaches on the link prediction task over  $A_{test}$  when overlapping arc types are not considered. As expected, the superiority of the subsampling approach is unaffected. Instead, it is interesting to notice the effects of overlapping relations on the arc prediction task. The removal of overlapping arc types seems to affect positively the metrics related to the task of *Edge Existence*. This is reasonable, as the arc existence task compresses the knowledge of arc type to its mere existence. Therefore, overlapping relations might confuse the model when considering predicting only existence of an arc.

On the other hand, metrics related to the task of classifying arc type – i.e. *Edge Class* – seem

| Approach           | Edge Existence |       |          | Edge Class |                |       |
|--------------------|----------------|-------|----------|------------|----------------|-------|
|                    | AUC            | AP    | Accuracy | EMR        | F <sub>1</sub> | PEA   |
| <i>Plain</i>       | 0.857          | 0.857 | 0.857    | 0.286      | 0.214          | 0.738 |
| <i>Subsampling</i> | 0.949          | 0.961 | 0.938    | 0.670      | 0.340          | 0.935 |

**Table 2**

Performance of the two different approaches over  $A_{test}$  when overlapping arc types – e.g. *father* and *parent* – are not considered.

to remain more or less untouched. This is a positive signal, as it indicates that GNNs can handle well overlapping relations between entities, without incurring in degradation of performance. However, further study of the behaviour of the proposed model in the presence of overlapping or non-overlapping relations is required for future extensions.

## 5. Conclusions

In this paper we propose a theoretical framework that leverage translation techniques between logic theories and graphs so as to tackle relevant logical problems. We identify the relevant logical problems and describe how they can benefit from the use of sub-symbolical approaches. Along with our framework, we identify possible mapping functions between logic and graphs, stressing their required properties and defining how these should be identified by users. We then consider the filling of a fragmented theory as a study case and apply the proposed framework to this task. Obtained results show the goodness of our approach, introducing the possibility to leverage GNN and graph ML techniques to identify missing latent relations between entities of a theory. Finally, a brief study on GNNs behaviour for overlapping classes in link prediction problem is presented, showing GNNs reliability, as well as potential for future development and more detailed study of the phenomena.

Future works should focus on applying our framework for tackling the many relevant logical tasks that exists in the CL world. Moreover, we consider relevant for future works to focus on some limitations of GNNs, which we discovered while working on our framework. Indeed, many logical tasks require mapping of predicates having arity greater than two to hypergraphs. Furthermore, some logical tasks require considering high number of relations, which are mapped to highly dimensional arc vectors. State-of-the-art GNNs still suffer these requirements, especially if combined. We believe that future research in GNNs should propose models capable of working on complex hypergraphs characterised by high dimensional arc features. Finally, given the centrality role of arc relations in our framework, it would be desirable to propose GNN models more focused on the relevance of arc attributes. Indeed, information is updated only at vertices level – i.e.  $\mathbf{x}_v^t$  – in GNNs. However, we believe that logical tasks, and GNNs performance more in general, may benefit from updating arc information as well—e.g.  $\mathbf{a}_{v,w}^t$ . Few works have been proposed to tackle this issue [21, 22], which are characterised by strong requirements and poor generalisability. Therefore, we believe it exists further research potential to develop GNNs leveraging arc information more efficiently.

## Acknowledgments

This paper has been partially supported by (i) the H2020 project “StairwAI” (G.A. 101017142), and (ii) the CHIST-ERA IV project “EXPECTATION” (G.A. CHIST-ERA-19-XAI-005).

## References

- [1] A. B. Arrieta, N. D. Rodríguez, J. D. Ser, A. Bennetot, S. Tabik, A. Barbado, S. García, S. Gil-Lopez, D. Molina, R. Benjamins, R. Chatila, F. Herrera, Explainable artificial intelligence (XAI): concepts, taxonomies, opportunities and challenges toward responsible AI, *Information Fusion* 58 (2020) 82–115. doi:10.1016/j.inffus.2019.12.012.
- [2] R. Calegari, G. Ciatto, A. Omicini, On the integration of symbolic and sub-symbolic techniques for XAI: A survey, *Intelligenza Artificiale* 14 (2020) 7–32. doi:10.3233/IA-190036.
- [3] F. J. Kurfess, Integrating symbol-oriented and sub-symbolic reasoning methods into hybrid systems, in: *From Synapses to Rules*, Springer, 2002, pp. 275–292.
- [4] M. L. Minsky, Logical versus analogical or symbolic versus connectionist or neat versus scruffy, *AI Magazine* 12 (1991) 34. doi:10.1609/aimag.v12i2.894.
- [5] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, P. S. Yu, A comprehensive survey on graph neural networks, *IEEE Transactions on Neural Networks Learning Systems* 32 (2021) 4–24. doi:10.1109/TNNLS.2020.2978386.
- [6] Z. Shui, G. Karypis, Heterogeneous molecular graph neural networks for predicting molecule properties, in: C. Plant, H. Wang, A. Cuzzocrea, C. Zaniolo, X. Wu (Eds.), 20th IEEE International Conference on Data Mining, ICDM 2020, Sorrento, Italy, November 17-20, 2020, IEEE, 2020, pp. 492–500. doi:10.1109/ICDM50108.2020.00058.
- [7] H. Wang, J. Leskovec, Unifying graph convolutional neural networks and label propagation, *CoRR abs/2002.06755* (2020). arXiv:2002.06755.
- [8] W. Fan, Y. Ma, Q. Li, Y. He, Y. E. Zhao, J. Tang, D. Yin, Graph neural networks for social recommendation, in: L. Liu, R. W. White, A. Mantrach, F. Silvestri, J. J. McAuley, R. Baeza-Yates, L. Zia (Eds.), *The World Wide Web Conference, WWW 2019, San Francisco, CA, USA, May 13-17, 2019*, ACM, 2019, pp. 417–426. doi:10.1145/3308558.3313488.
- [9] R. Hecht-Nielsen, Theory of the backpropagation neural network, *Neural Networks* 1 (1988) 445–448. doi:10.1016/0893-6080(88)90469-8.
- [10] L. C. Paulson, Computational logic: its origins and applications, *Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences* 474 (2018) 20170872. doi:10.1098/rspa.2017.0872.
- [11] T. Eiter, M. Fink, H. Tompits, S. Woltran, Simplifying logic programs under uniform and strong equivalence, in: V. Lifschitz, I. Niemelä (Eds.), *Logic Programming and Nonmonotonic Reasoning, 7th International Conference, LPNMR 2004, Fort Lauderdale, FL, USA, January 6-8, 2004*, Proceedings, volume 2923 of *Lecture Notes in Computer Science*, Springer, 2004, pp. 87–99. doi:10.1007/978-3-540-24609-1\_10.
- [12] J. Ji, H. Wan, Z. Huo, Z. Yuan, Simplifying A logic program using its consequences, in: Q. Yang, M. J. Wooldridge (Eds.), *Proceedings of the Twenty-Fourth International Joint*



- Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015, AAAI Press, 2015, pp. 3069–3075. URL: <http://ijcai.org/Abstract/15/433>.
- [13] S. Bai, F. Zhang, P. H. S. Torr, Hypergraph convolution and hypergraph attention, *Pattern Recognition* 110 (2021) 107637. doi:10.1016/j.patcog.2020.107637.
- [14] J. E. Shore, R. W. Johnson, Properties of cross-entropy minimization, *IEEE Transactions on Information Theory* 27 (1981) 472–482. doi:10.1109/TIT.1981.1056373.
- [15] M. Zhang, Y. Chen, Link prediction based on graph neural networks, in: S. Bengio, H. M. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, R. Garnett (Eds.), *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada, 2018*, pp. 5171–5181. URL: <https://proceedings.neurips.cc/paper/2018/hash/53f0d7c537d99b3824f0f99d62ea2428-Abstract.html>.
- [16] M. Fey, J. E. Lenssen, Fast graph representation learning with pytorch geometric, *CoRR abs/1903.02428* (2019). arXiv:1903.02428.
- [17] M. Wang, L. Yu, D. Zheng, Q. Gan, Y. Gai, Z. Ye, M. Li, J. Zhou, Q. Huang, C. Ma, Z. Huang, Q. Guo, H. Zhang, H. Lin, J. Zhao, J. Li, A. J. Smola, Z. Zhang, Deep graph library: Towards efficient and scalable deep learning on graphs, *CoRR abs/1909.01315* (2019). arXiv:1909.01315.
- [18] J. Wang, Y. Yang, J. Mao, Z. Huang, C. Huang, W. Xu, CNN-RNN: A unified framework for multi-label image classification, in: *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*, IEEE Computer Society, 2016, pp. 2285–2294. doi:10.1109/CVPR.2016.251.
- [19] J. A. Hanley, B. J. McNeil, The meaning and use of the area under a receiver operating characteristic (roc) curve., *Radiology* 143 (1982) 29–36.
- [20] A. Clauset, C. Moore, M. E. J. Newman, Hierarchical structure and the prediction of missing links in networks, *Nature* 453 (2008) 98–101.
- [21] L. Gong, Q. Cheng, Exploiting edge features for graph neural networks, in: *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2019, Long Beach, CA, USA, June 16-20, 2019*, Computer Vision Foundation / IEEE, 2019, pp. 9211–9219. doi:10.1109/CVPR.2019.00943.
- [22] X. Jiang, R. Zhu, P. Ji, S. Li, Co-embedding of nodes and edges with graph neural networks, *CoRR abs/2010.13242* (2020). arXiv:2010.13242.