GATED GRAPH SEQUENCE NEURAL NETWORKS

Yujia Li[*] **& Richard Zemel**
Department of Computer Science, University of Toronto
Toronto, Canada
{yujiali,zemel}@cs.toronto.edu

Marc Brockschmidt & Daniel Tarlow
Microsoft Research
Cambridge, UK
{mabrocks,dtarlow}@microsoft.com

ABSTRACT

Graph-structured data appears frequently in domains including chemistry, natural language semantics, social networks, and knowledge bases. In this work, we study feature learning techniques for graph-structured inputs. Our starting point is previous work on Graph Neural Networks (Scarselli et al., 2009), which we modify to use gated recurrent units and modern optimization techniques and then extend to output sequences. The result is a flexible and broadly useful class of neural network models that has favorable inductive biases relative to purely sequence-based models (e.g., LSTMs) when the problem is graph-structured. We demonstrate the capabilities on some simple AI (bAbI) and graph algorithm learning tasks. We then show it achieves state-of-the-art performance on a problem from program verification, in which subgraphs need to be described as abstract data structures.

1 INTRODUCTION

Many practical applications build on graph-structured data, and thus we often want to perform machine learning tasks that take graphs as inputs. Standard approaches to the problem include engineering custom features of an input graph, graph kernels (Kashima et al., 2003; Shervashidze et al., 2011), and methods that define graph features in terms of random walks on graphs (Perozzi et al., 2014). More closely related to our goal in this work are methods that learn features on graphs, including Graph Neural Networks (Gori et al., 2005; Scarselli et al., 2009), spectral networks (Bruna et al., 2013) and recent work on learning graph fingerprints for classification tasks on graph representations of chemical molecules (Duvenaud et al., 2015).

Our main contribution is an extension of Graph Neural Networks that outputs sequences. Previous work on feature learning for graph-structured inputs has focused on models that produce single outputs such as graph-level classifications, but many problems with graph inputs require outputting sequences. Examples include paths on a graph, enumerations of graph nodes with desirable properties, or sequences of global classifications mixed with, for example, a start and end node. We are not aware of existing graph feature learning work suitable for this problem. Our motivating application comes from program verification and requires outputting logical formulas, which we formulate as a sequential output problem. A secondary contribution is highlighting that Graph Neural Networks (and further extensions we develop here) are a broadly useful class of neural network model that is applicable to many problems currently facing the field.

There are two settings for feature learning on graphs: (1) learning a representation of the input graph, and (2) learning representations of the internal state during the process of producing a sequence of outputs. Here, (1) is mostly achieved by previous work on Graph Neural Networks (Scarselli et al., 2009); we make several minor adaptations of this framework, including changing it to use modern practices around Recurrent Neural Networks. (2) is important because we desire outputs from graph-structured problems that are not solely individual classifications. In these cases, the challenge is how

[*]Work done primarily while author was an intern at Microsoft Research.

to learn features on the graph that encode the partial output sequence that has already been produced (e.g., the path so far if outputting a path) and that still needs to be produced (e.g., the remaining path). We will show how the GNN framework can be adapted to these settings, leading to a novel graph-based neural network model that we call Gated Graph Sequence Neural Networks (GGS-NNs).

We illustrate aspects of this general model in experiments on bAbI tasks (Weston et al., 2015) and graph algorithm learning tasks that illustrate the capabilities of the model. We then present an application to the verification of computer programs. When attempting to prove properties such as *memory safety* (i.e., that there are no null pointer dereferences in a program), a core problem is to find mathematical descriptions of the data structures used in a program. Following Brockschmidt et al. (2015), we have phrased this as a machine learning problem where we will learn to map from a set of input graphs, representing the state of memory, to a logical description of the data structures that have been instantiated. Whereas Brockschmidt et al. (2015) relied on a large amount of hand-engineering of features, we show that the system can be replaced with a GGS-NN at no cost in accuracy.

2 Graph Neural Networks

In this section, we review Graph Neural Networks (GNNs) (Gori et al., 2005; Scarselli et al., 2009) and introduce notation and concepts that will be used throughout.

GNNs are a general neural network architecture defined according to a graph structure $\mathcal{G} = (\mathcal{V}, \mathcal{E})$. Nodes $v \in \mathcal{V}$ take unique values from $1, \ldots, |\mathcal{V}|$, and edges are pairs $e = (v, v') \in \mathcal{V} \times \mathcal{V}$. We will focus in this work on directed graphs, so (v, v') represents a directed edge $v \to v'$, but we note that the framework can easily be adapted to undirected graphs; see Scarselli et al. (2009). The *node vector* (or *node representation* or *node embedding*) for node v is denoted by $\mathbf{h}_v \in \mathbb{R}^D$. Graphs may also contain node labels $l_v \in \{1, \ldots, L_\mathcal{V}\}$ for each node v and edge labels or edge types $l_e \in \{1, \ldots, L_\mathcal{E}\}$ for each edge. We will overload notation and let $\mathbf{h}_\mathcal{S} = \{\mathbf{h}_v \mid v \in \mathcal{S}\}$ when $\mathcal{S}$ is a set of nodes, and $l_\mathcal{S} = \{l_e \mid e \in \mathcal{S}\}$ when $\mathcal{S}$ is a set of edges. The function $\textsc{In}(v) = \{v' \mid (v', v) \in \mathcal{E}\}$ returns the set of predecessor nodes v' with $v' \to v$. Analogously, $\textsc{Out}(v) = \{v' \mid (v, v') \in \mathcal{E}\}$ is the set of successor nodes v' with edges $v \to v'$. The set of all nodes neighboring v is $\textsc{Nbr}(v) = \textsc{In}(v) \cup \textsc{Out}(v)$, and the set of all edges incoming to or outgoing from v is $\textsc{Co}(v) = \{(v', v'') \in \mathcal{E} \mid v = v' \vee v = v''\}$.

GNNs map graphs to outputs via two steps. First, there is a propagation step that computes node representations for each node; second, an output model $o_v = g(\mathbf{h}_v, l_v)$ maps from node representations and corresponding labels to an output o_v for each $v \in \mathcal{V}$. In the notation for g, we leave the dependence on parameters implicit, and we will continue to do this throughout. The system is differentiable from end-to-end, so all parameters are learned jointly using gradient-based optimization.

2.1 Propagation Model

Here, an iterative procedure propagates node representations. Initial node representations $\mathbf{h}_v^{(1)}$ are set to arbitrary values, then each node representation is updated following the recurrence below until convergence, where t denotes the timestep:

$$\mathbf{h}_v^{(t)} = f^*(l_v, l_{\textsc{Co}(v)}, l_{\textsc{Nbr}(v)}, \mathbf{h}_{\textsc{Nbr}(v)}^{(t-1)}).$$

Several variants are discussed in Scarselli et al. (2009) including positional graph forms, node-specific updates, and alternative representations of neighborhoods. Concretely, Scarselli et al. (2009) suggest decomposing $f^*(\cdot)$ to be a sum of per-edge terms:

$$f^*(l_v, l_{\textsc{Co}(v)}, l_{\textsc{Nbr}(v)}, \mathbf{h}_{\textsc{Nbr}(v)}^{(t)}) = \sum_{v' \in \textsc{In}(v)} f(l_v, l_{(v',v)}, l_{v'}, \mathbf{h}_{v'}^{(t-1)}) + \sum_{v' \in \textsc{Out}(v)} f(l_v, l_{(v,v')}, l_{v'}, \mathbf{h}_{v'}^{(t-1)}),$$

where $f(\cdot)$ is either a linear function of $\mathbf{h}_{v'}$ or a neural network. The parameters of f depends on the configuration of labels, e.g. in the following linear case, $\mathbf{A}$ and $\mathbf{b}$ are learnable parameters,

$$f(l_v, l_{(v',v)}, l_{v'}, \mathbf{h}_{v'}^{(t)}) = \mathbf{A}^{(l_v, l_{(v',v)}, l_{v'})} \mathbf{h}_{v'}^{(t-1)} + \mathbf{b}^{(l_v, l_{(v',v)}, l_{v'})}.$$

2.2 Output Model and Learning

The output model is defined per node and is a differentiable function $g(\mathbf{h}_v, l_v)$ that maps to an output. This is generally a linear or neural network mapping. Scarselli et al. (2009) focus on outputs that are independent per node, which are implemented by mapping the final node representations $\mathbf{h}_v^{(T)}$, to an output $o_v = g(\mathbf{h}_v^{(T)}, l_v)$ for each node $v \in \mathcal{V}$. To handle graph-level classifications, they suggest to create a dummy "super node" that is connected to all other nodes by a special type of edge. Thus, graph-level regression or classification can be handled in the same manner as node-level regression or classification.

Learning is done via the Almeida-Pineda algorithm (Almeida, 1990; Pineda, 1987), which works by running the propagation to convergence, and then computing gradients based upon the converged solution. This has the advantage of not needing to store intermediate states in order to compute gradients. The disadvantage is that parameters must be constrained so that the propagation step is a contraction map. This is needed to ensure convergence, but it may limit the expressivity of the model. When $f(\cdot)$ is a neural network, this is encouraged using a penalty term on the 1-norm of the network's Jacobian. See Appendix A for an example that gives the intuition that contraction maps have trouble propagating information across a long range in a graph.

3 Gated Graph Neural Networks

We now describe Gated Graph Neural Networks (GG-NNs), our adaptation of GNNs that is suitable for non-sequential outputs. We will describe sequential outputs in the next section. The biggest modification of GNNs is that we use Gated Recurrent Units (Cho et al., 2014) and unroll the recurrence for a fixed number of steps T and use backpropagation through time in order to compute gradients. This requires more memory than the Almeida-Pineda algorithm, but it removes the need to constrain parameters to ensure convergence. We also extend the underlying representations and output model.

3.1 Node Annotations

In GNNs, there is no point in initializing node representations because the contraction map constraint ensures that the fixed point is independent of the initializations. This is no longer the case with GG-NNs, which lets us incorporate node labels as additional inputs. To distinguish these node labels used as inputs from the ones introduced before, we call them *node annotations*, and use vector $\boldsymbol{x}$ to denote these annotations.

To illustrate how the node annotations are used, consider an example task of training a graph neural network to predict whether node t can be reached from node s on a given graph. For this task, there are two problem-related special nodes, s and t. To mark these nodes as special, we give them an initial annotation. The first node s gets the annotation $\boldsymbol{x}_s = [1, 0]^\top$, and the second node t gets the annotation $\boldsymbol{x}_t = [0, 1]^\top$. All other nodes v have their initial annotation set to $\boldsymbol{x}_v = [0, 0]^\top$. Intuitively, this marks s as the first input argument and t as the second input argument. We then initialize the node state vectors $\mathbf{h}_v^{(1)}$ using these label vectors by copying $\boldsymbol{x}_v$ into the first dimensions and padding with extra 0's to allow hidden states that are larger than the annotation size.

In the reachability example, it is easy for the propagation model to learn to propagate the node annotation for s to all nodes reachable from s, for example by setting the propagation matrix associated with forward edges to have a 1 in position (0,0). This will cause the first dimension of node representation to be copied along forward edges. With this setting of parameters, the propagation step will cause all nodes reachable from s to have their first bit of node representation set to 1. The output step classifier can then easily tell whether node t is reachable from s by looking whether some node has nonzero entries in the first two dimensions of its representation vector.

3.2 Propagation Model

The basic recurrence of the propagation model is

$$\mathbf{h}_v^{(1)} = [\boldsymbol{x}_v^\top, \mathbf{0}]^\top \tag{1}$$

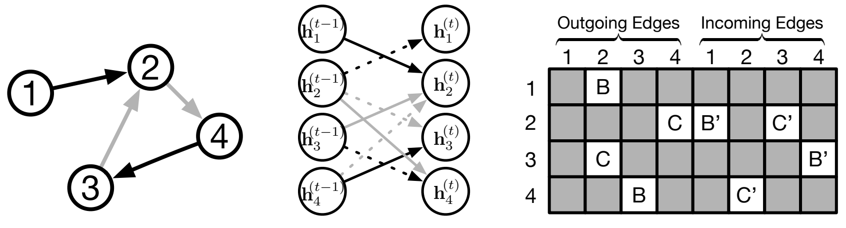
$$\mathbf{a}_v^{(t)} = \mathbf{A}_{v:}^\top \left[\mathbf{h}_1^{(t-1)\top} \dots \mathbf{h}_{|\mathcal{V}|}^{(t-1)\top}\right]^\top + \mathbf{b} \tag{2}$$

$$\mathbf{z}_v^t = \sigma\left(\mathbf{W}^z \mathbf{a}_v^{(t)} + \mathbf{U}^z \mathbf{h}_v^{(t-1)}\right) \tag{3}$$

$$\mathbf{r}_v^t = \sigma\left(\mathbf{W}^r \mathbf{a}_v^{(t)} + \mathbf{U}^r \mathbf{h}_v^{(t-1)}\right) \tag{4}$$

$$\widetilde{\mathbf{h}_v^{(t)}} = \tanh\left(\mathbf{W}\mathbf{a}_v^{(t)} + \mathbf{U}\left(\mathbf{r}_v^t \odot \mathbf{h}_v^{(t-1)}\right)\right) \tag{5}$$

$$\mathbf{h}_v^{(t)} = (1 - \mathbf{z}_v^t) \odot \mathbf{h}_v^{(t-1)} + \mathbf{z}_v^t \odot \widetilde{\mathbf{h}_v^{(t)}}. \tag{6}$$



(a) (b) (c) $\mathbf{A} = \left[\mathbf{A}^{(\text{out})}, \mathbf{A}^{(\text{in})}\right]$

Figure 1: (a) Example graph. Color denotes edge types. (b) Unrolled one timestep. (c) Parameter tying and sparsity in recurrent matrix. Letters denote edge types with B' corresponding to the reverse edge of type B. B and B' denote distinct parameters.

The matrix $\mathbf{A} \in \mathbb{R}^{D|\mathcal{V}| \times 2D|\mathcal{V}|}$ determines how nodes in the graph communicate with each other. The sparsity structure and parameter tying in $\mathbf{A}$ is illustrated in Fig. 1. The sparsity structure corresponds to the edges of the graph, and the parameters in each submatrix are determined by the edge type and direction. $\mathbf{A}_{v:} \in \mathbb{R}^{D \times 2D|\mathcal{V}|}$ is the submatrix of $\mathbf{A}$ containing the rows corresponding to node v. Eq. 1 is the initialization step, which copies node annotations into the first components of the hidden state and pads the rest with zeros. Eq. 2 is the step that passes information between different nodes of the graph via incoming and outgoing edges with parameters dependent on the edge type and direction. $\mathbf{a}_v^{(t)} \in \mathbb{R}^{2D}$ contains activations from edges in both directions. The remaining are GRU-like updates that incorporate information from the other nodes and from the previous timestep to update each node's hidden state. $\mathbf{z}$ and $\mathbf{r}$ are the update and reset gates, $\sigma(x) = 1/(1+e^{-x})$ is the logistic sigmoid function, and $\odot$ is element-wise multiplication. We initially experimented with a vanilla recurrent neural network-style update, but in preliminary experiments we found this GRU-like propagation step to be more effective.

3.3 Output Models

There are several types of one-step outputs that we would like to produce in different situations. First, GG-NNs support *node selection* tasks by making $o_v = g(\mathbf{h}_v^{(T)}, \boldsymbol{x}_v)$ for each node $v \in \mathcal{V}$ output node scores and applying a softmax over node scores. Second, for graph-level outputs, we define a graph level representation vector as

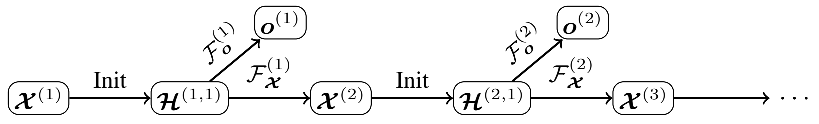
$$\mathbf{h}_{\mathcal{G}} = \tanh\left(\sum_{v \in \mathcal{V}} \sigma\left(i(\mathbf{h}_v^{(T)}, \boldsymbol{x}_v)\right) \odot \tanh\left(j(\mathbf{h}_v^{(T)}, \boldsymbol{x}_v)\right)\right), \tag{7}$$

where $\sigma(i(\mathbf{h}_v^{(T)}, \boldsymbol{x}_v))$ acts as a soft attention mechanism that decides which nodes are relevant to the current graph-level task. i and j are neural networks that take the concatenation of $\mathbf{h}_v^{(T)}$ and $\boldsymbol{x}_v$ as input and outputs real-valued vectors. The $\tanh$ functions can also be replaced with the identity.

4 Gated Graph Sequence Neural Networks

Here we describe Gated Graph Sequence Neural Networks (GGS-NNs), in which several GG-NNs operate in sequence to produce an output sequence $\boldsymbol{o}^{(1)} \ldots \boldsymbol{o}^{(K)}$.

For the k^{th} output step, we denote the matrix of node annotations as $\boldsymbol{\mathcal{X}}^{(k)} = [\boldsymbol{x}_1^{(k)}; \ldots; \boldsymbol{x}_{|\mathcal{V}|}^{(k)}]^\top \in \mathbb{R}^{|\mathcal{V}| \times L_{\mathcal{V}}}$. We use two GG-NNs $\mathcal{F}_{\boldsymbol{o}}^{(k)}$ and $\mathcal{F}_{\boldsymbol{\mathcal{X}}}^{(k)}$: $\mathcal{F}_{\boldsymbol{o}}^{(k)}$ for predicting $\boldsymbol{o}^{(k)}$ from $\boldsymbol{\mathcal{X}}^{(k)}$, and $\mathcal{F}_{\boldsymbol{\mathcal{X}}}^{(k)}$ for predicting $\boldsymbol{\mathcal{X}}^{(k+1)}$ from $\boldsymbol{\mathcal{X}}^{(k)}$. $\boldsymbol{\mathcal{X}}^{(k+1)}$ can be seen as the states carried over from step k to $k+1$. Both $\mathcal{F}_{\boldsymbol{o}}^{(k)}$ and $\mathcal{F}_{\boldsymbol{\mathcal{X}}}^{(k)}$ contain a propagation model and an output model. In the propagation models, we denote the matrix of node vectors at the t^{th} propagation step of the k^{th} output step as $\boldsymbol{\mathcal{H}}^{(k,t)} = [\mathbf{h}_1^{(k,t)}; \ldots; \mathbf{h}_{|\mathcal{V}|}^{(k,t)}]^\top \in \mathbb{R}^{|\mathcal{V}| \times D}$. As before, in step k, we set $\boldsymbol{\mathcal{H}}^{(k,1)}$ by 0-extending $\boldsymbol{\mathcal{X}}^{(k)}$ per node. An overview of the model is shown in Fig. 2. Alternatively, $\mathcal{F}_{\boldsymbol{o}}^{(k)}$ and $\mathcal{F}_{\boldsymbol{\mathcal{X}}}^{(k)}$ can share a single propagation model, and just have separate output models. This simpler variant is faster to train and evaluate, and



$$\begin{tikzcd} & \boxed{\boldsymbol{o}^{(1)}} & & & \boxed{\boldsymbol{o}^{(2)}} & & \\ \boxed{\boldsymbol{\mathcal{X}}^{(1)}} \arrow[r, "\text{Init}"] & \boxed{\boldsymbol{\mathcal{H}}^{(1,1)}} \arrow[u, "\mathcal{F}_{\boldsymbol{o}}^{(1)}"] \arrow[r, "\mathcal{F}_{\boldsymbol{\mathcal{X}}}^{(1)}"] & \boxed{\boldsymbol{\mathcal{X}}^{(2)}} \arrow[r, "\text{Init}"] & \boxed{\boldsymbol{\mathcal{H}}^{(2,1)}} \arrow[u, "\mathcal{F}_{\boldsymbol{o}}^{(2)}"] \arrow[r, "\mathcal{F}_{\boldsymbol{\mathcal{X}}}^{(2)}"] & \boxed{\boldsymbol{\mathcal{X}}^{(3)}} \arrow[r] & \cdots \end{tikzcd}$$

Figure 2: Architecture of GGS-NN models.

in many cases can achieve similar performance level as the full model. But in cases where the desired propagation behavior for $\mathcal{F}_{\boldsymbol{o}}^{(k)}$ and $\mathcal{F}_{\boldsymbol{\mathcal{X}}}^{(k)}$ are different, this variant may not work as well.

We introduce a *node annotation* output model for predicting $\boldsymbol{\mathcal{X}}^{(k+1)}$ from $\boldsymbol{\mathcal{H}}^{(k,T)}$. The prediction is done for each node independently using a neural network $j(\mathbf{h}_v^{(k,T)}, \boldsymbol{x}_v^{(k)})$ that takes the concatenation of $\mathbf{h}_v^{(k,T)}$ and $\boldsymbol{x}_v^{(k)}$ as input and outputs a vector of real-valued scores:

$$\boldsymbol{x}_v^{(k+1)} = \sigma\left(j(\mathbf{h}_v^{(k,T)}, \boldsymbol{x}_v^{(k)})\right). \tag{8}$$

There are two settings for training GGS-NNs: specifying all intermediate annotations $\boldsymbol{\mathcal{X}}^{(k)}$, or training the full model end-to-end given only $\boldsymbol{\mathcal{X}}^{(1)}$, graphs and target sequences. The former can improve performance when we have domain knowledge about specific intermediate information that should be represented in the internal state of nodes, while the latter is more general. We describe both.

Sequence outputs with observed annotations Consider the task of making a sequence of predictions for a graph, where each prediction is only about a part of the graph. In order to ensure we predict an output for each part of the graph exactly once, it suffices to have one bit per node, indicating whether the node has been "explained" so far. In some settings, a small number of annotations are sufficient to capture the state of the output procedure. When this is the case, we may want to directly input this information into the model via labels indicating target intermediate annotations. In some cases, these annotations may be *sufficient*, in that we can define a model where the GG-NNs are rendered conditionally independent given the annotations.

In this case, at training time, given the annotations $\boldsymbol{\mathcal{X}}^{(k)}$ the sequence prediction task decomposes into single step prediction tasks and can be trained as separate GG-NNs. At test time, predicted annotations from one step will be used as input to the next step. This is analogous to training directed graphical models when data is fully observed.

Sequence outputs with latent annotations More generally, when intermediate node annotations $\boldsymbol{\mathcal{X}}^{(k)}$ are not available during training, we treat them as hidden units in the network, and train the whole model jointly by backpropagating through the whole sequence.

5 Explanatory Applications

In this section we present example applications that concretely illustrate the use of GGS-NNs. We focus on a selection of bAbI artificial intelligence (AI) tasks (Weston et al., 2015) and two graph algorithm learning tasks.

5.1 bAbI Tasks

The bAbI tasks are meant to test reasoning capabilities that AI systems should be capable of. In the bAbI suite, there are 20 tasks that test basic forms of reasoning like deduction, induction, counting, and path-finding.

We have defined a basic transformation procedure that maps bAbI tasks to GG-NNs or GGS-NNs. We use the `--symbolic` option from the released bAbI code to get stories that just involve sequences of relations between entities, which are then converted into a graph. Each entity is mapped to a node, and each relation is mapped to an edge with edge label given by the relation. The full story is consumed and mapped to a single graph. Questions are marked by `eval` in the data and are comprised of a question type (e.g., `has_fear`), and some argument (e.g., one or more nodes). The arguments are converted into initial node annotations, with the i-th bit of the i-th argument node's annotation vector set to 1. For example, if the `eval` line is `eval E > A true`, then `E` gets initial annotation $\boldsymbol{x}_E^{(1)} = [1, 0]^\top$, `A` gets $\boldsymbol{x}_A^{(1)} = [0, 1]^\top$, and for all other nodes v, $\boldsymbol{x}_v^{(1)} = [0, 0]^\top$. Question type is 1 (for '>') and output is class 1 (for 'true'). Some tasks have multiple question types, for example Task 4 which has 4 question types: `e`, `s`, `w`, `n`. For such tasks we simply train a separate GG-NN for each task. We do not use the strong supervision labels or give the GGS-NNs any intermediate annotations in any experiments.

While simple, this transformation does not preserve all information about the story (e.g., it discards temporal order of the inputs), and it does not easily handle ternary and higher order relations (e.g., `Yesterday John went to the garden` is not easily mapped to a simple edge). We also emphasize that it is a non-trivial task to map general natural language to symbolic form,[1] so we could not directly apply this approach to arbitrary natural language. Relaxing these restrictions is left for future work.

However, even with this simple transformation, there are a variety of bAbI tasks that can be formulated, including Task 19 (Path Finding), which is arguably the hardest task. We provide baselines to show that the symbolic representation does not help RNNs or LSTMs significantly, and show that GGS-NNs solve the problem with a small number of training instances. We also develop two new bAbI-like tasks that involve outputting sequences on graphs: shortest paths, and a simple form of Eulerian circuits (on random connected 2-regular graphs). The point of these experiments is to illustrate the capabilities of GGS-NNs across a variety of problems.

Example 1. As an example, below is an instance from the symbolic dataset for bAbI task 15, Basic Deduction.

```
D is A
B is E
A has_fear F
G is F
E has_fear H
F has_fear A
H has_fear A
C is H
eval B has_fear       H
eval G has_fear       A
eval C has_fear       A
eval D has_fear       F
```

Here the first 8 lines describe the facts, the GG-NN will use these facts to build a graph. Capital letters are nodes, `is` and `has_fear` are interpreted as edge labels or edge types. The last 4 lines are 4 questions asked for this input data. `has_fear` in these lines are interpreted as a question type. For this task, in each question only one node is special, e.g. the `B` in `eval B has_fear`, and we assign a single value 1 to the annotation vector for this special node and 0 to all the other nodes.

For RNN and LSTM the data is converted into token sequences like below:

```
n6 e1 n1 eol n6 e1 n5 eol n1 e1 n2 eol n4 e1 n5 eol n3 e1 n4
eol n3 e1 n5 eol n6 e1 n4 eol q1 n6 n2 ans 1
```

where `n<id>` are nodes, `e<id>` are edges, `q<id>` are question types, extra tokens `eol` (end-of-line) and `ans` (answer) are added to give the RNN & LSTM access to the complete information available in the dataset. The final number is the class label.

[1]Although the bAbI data is quite templatic, so it is straightforward to hand-code a parser that will work for the bAbI data; the symbolic option removes the need for this.

Example 2. As a second example, below is an instance from the symbolic dataset for bAbI task 19, Path Finding.

```
E s A
B n C
E w F
B w E
eval path B A w,s
```

Here the first 4 lines describe edges, `s`, `n`, `w`, `e` (`e` does not appear in this example) are all different edge types. The last line is a path question, the answer is a sequence of directions `w,s`, as the path going from `B` to `A` is to first go west to `E` then go south to `A`. The `s`, `n`, `w`, `e` in the question lines are treated as output classes.

More Training Details. For all tasks in this section, we generate 1000 training examples and 1000 test examples, 50 of the training examples are used for validation. When evaluating model performance, for all bAbI tasks that contain more than one questions in one example, the predictions for different questions were evaluated independently. As there is randomness in the dataset generation process, we generated 10 such datasets for each task, and report the mean and standard deviation of the evaluation performance across the 10 datasets.

For all explanatory tasks, we start by training different models on only 50 training examples, and gradually increase the number of training examples to 100, 250, 500, and 950 (50 of the training examples are reserved for validation) until the model's test accuracy reaches 95% or above, a success by bAbI standard Weston et al. (2015). For each method, we report the minimum number of training examples it needs to reach 95% accuracy along with the accuracy it reaches with that amount of training examples. In all these cases, we unrolled the propagation process for 5 steps. For bAbI task 4, 15, 16, 18, 19, we used GG-NN with the size of node vectors $\mathbf{h}_v^{(t)}$ set to $D = 4$, $D = 5$, $D = 6$, $D = 3$ and $D = 6$ respectively. For all the GGS-NNs in this section we used the simpler variant in which $\mathcal{F}_{\boldsymbol{o}}^{(k)}$ and $\mathcal{F}_{\boldsymbol{\mathcal{X}}}^{(k)}$ share a single propagation model. For shortest path and Eulerian circuit tasks, we used $D = 20$. All models are trained long enough with Adam (Kingma & Ba, 2014), and the validation set is used to choose the best model to evaluate and avoid models that are overfitting.

5.1.1 Single Step Outputs

We choose four bAbI tasks that are suited to the restrictions described above and require single step outputs: 4 (Two Argument Relations), 15 (Basic Deduction), 16 (Basic Induction), and 18 (Size Reasoning). For Task 4, 15 and 16, a node selection GG-NN is used. For Task 18 we used a graph-level classification version. All the GGNN networks contain less than 600 parameters[2].

As baselines, we train RNN and LSTM models on the symbolic data in raw sequence form. The RNNs and LSTMs use 50 dimensional embeddings and 50 dimensional hidden layers; they predict a single output at the end of the sequences and the output is treated as a classification problem, the loss is cross entropy. The RNNs and LSTMs contain around 5k and 30k parameters, respectively.

Test results appear in Table 1. For all tasks GG-NN achieves perfect test accuracy using only 50 training examples, while the RNN/LSTM baselines either use more training examples (Task 4) or fail to solve the tasks (Task 15, 16 and 18).

In Table 2, we further break down performance of the baselines for task 4 as the amount of training data varies. While both the RNN and LSTM are able to solve the task almost perfectly, the GG-NN reaches 100% accuracy with much less data.

[2]For bAbI task 4, we treated 'e', 's', 'w', 'n' as 4 question types and trained one GG-NN for each question type, so strictly speaking for bAbI task 4 our GG-NN model has 4 times the number of parameters of a single GG-NN model. In our experiments we used a GG-NN with 271 parameters for each question type which means 1084 parameters in total.

Task	RNN	LSTM	GG-NN
bAbI Task 4	97.3±1.9 (250)	97.4±2.0 (250)	100.0±0.0 (50)
bAbI Task 15	48.6±1.9 (950)	50.3±1.3 (950)	100.0±0.0 (50)
bAbI Task 16	33.0±1.9 (950)	37.5±0.9 (950)	100.0±0.0 (50)
bAbI Task 18	88.9±0.9 (950)	88.9±0.8 (950)	100.0±0.0 (50)

Table 1: Accuracy in percentage of different models for different tasks. Number in parentheses is number of training examples required to reach shown accuracy.

#Training Examples	50	100	250	500	950
RNN	76.7±3.8	90.2±4.0	97.3±1.9	98.4±1.3	99.7±0.4
LSTM	73.5±5.2	86.4±3.8	97.4±2.0	99.2±0.8	99.6±0.8

Table 2: Performance breakdown of RNN and LSTM on bAbI task 4 as the amount of training data changes.

5.1.2 Sequential Outputs

The bAbI Task 19 (Path Finding) is arguably the hardest task among all bAbI tasks (see e.g., (Sukhbaatar et al., 2015), which reports an accuracy of less than 20% for all methods that do not use the strong supervision). We apply a GGS-NN to this problem, again on the symbolic form of the data (so results are not comparable to those in (Sukhbaatar et al., 2015)). An extra ‘end’ class is added to the end of each output sequence; at test time the network will keep making predictions until it predicts the ‘end’ class.

The results for this task are given in Table 3. Both RNN and LSTM fail on this task. However, with only 50 training examples, our GGS-NNs achieve much better test accuracy than RNN and LSTM.

5.2 Learning Graph Algorithms

Task	RNN	LSTM	GGS-NNs		
bAbI Task 19	24.7±2.7 (950)	28.2±1.3 (950)	71.1±14.7 (50)	92.5±5.9 (100)	99.0±1.1 (250)
Shortest Path	9.7±1.7 (950)	10.5±1.2 (950)	100.0± 0.0 (50)		
Eulerian Circuit	0.3±0.2 (950)	0.1±0.2 (950)	100.0± 0.0 (50)		

Table 3: Accuracy in percentage of different models for different tasks. The number in parentheses is number of training examples required to reach that level of accuracy.

We further developed two new bAbI-like tasks based on algorithmic problems on graphs: Shortest Paths, and Eulerian Circuits. For the first, we generate random graphs and produce a story that lists all edges in the graphs. Questions come from choosing two random nodes A and B and asking for the shortest path (expressed as a sequence of nodes) that connects the two chosen nodes. We constrain the data generation to only produce questions where there is a unique shortest path from A to B of length at least 2. For Eulerian circuits, we generate a random two-regular connected graph and a separate random distractor graph. The question gives two nodes A and B to start the circuit, then the question is to return the Eulerian circuit (again expressed as a sequence of nodes) on the given subgraph that starts by going from A to B. Results are shown in the Table 3. RNN and LSTM fail on both tasks, but GGS-NNs learns to make perfect predictions using only 50 training examples.

6 Program Verification with GGS-NNs

Our work on GGS-NNs is motivated by a practical application in program verification. A crucial step in automatic program verification is the inference of *program invariants*, which approximate the set of program states reachable in an execution. Finding invariants about data structures is an open problem. As an example, consider the simple `C` function on the right.

```
node* concat(node* a, node* b) {
  if (a == NULL) return b;
  node* cur = a;
  while (cur.next != NULL)
    cur = cur->next;
  cur->next = b;
  return a;                       }
```

To prove that this program indeed concatenates the two lists a and b and that all pointer dereferences are valid, we need to (mathematically) characterize the program's heap in each iteration of the loop. For this, we use *separation logic* (O'Hearn et al., 2001; Reynolds, 2002), which uses *inductive predicates* to describe abstract data structures. For example, a list segment is defined as $\mathsf{ls}(x, y) \equiv x = y \vee \exists v, n. \mathsf{ls}(n, y) * x \mapsto \{\texttt{val} : v, \texttt{next} : n\}$, where $x \mapsto \{\texttt{val} : v, \texttt{next} : n\}$ means that x points to a memory region that contains a structure with `val` and `next` fields whose values are in turn v and n. The $*$ connective is a conjunction as $\wedge$ in Boolean logic, but additionally requires that its operators refer to "separate" parts of the heap. Thus, $\mathsf{ls}(\texttt{cur}, \texttt{NULL})$ implies that `cur` is either `NULL`, or that it points to two values v, n on the heap, where n is described by ls again. The formula $\exists t. \mathsf{ls}(\texttt{a}, \texttt{cur}) * \mathsf{ls}(\texttt{cur}, \texttt{NULL}) * \mathsf{ls}(\texttt{b}, t)$ is an *invariant* of the loop (i.e., it holds when entering the loop, and after every iteration). Using it, we can prove that no program run will fail due to dereferencing an unallocated memory address (this property is called *memory safety*) and that the function indeed concatenates two lists using a Hoare-style verification scheme (Hoare, 1969).

The hardest part of this process is coming up with formulas that describe data structures, and this is where we propose to use machine learning. Given a program, we run it a few times and extract the state of memory (represented as a graph; see below) at relevant program locations, and then predict a separation logic formula. Static program analysis tools (e.g., (Piskac et al., 2014)) can check whether a candidate formula is sufficient to prove the desired properties (e.g., memory safety).

6.1 FORMALIZATION

Representing Heap State as a Graph As inputs we consider directed, possibly cyclic graphs representing the heap of a program. These graphs can be automatically constructed from a program's memory state. Each graph node v corresponds to an address in memory at which a sequence of pointers $v_0, \ldots, v_k$ is stored (we ignore non-pointer values in this work). Graph edges reflect these pointer values, i.e., v has edges labeled with $0, \ldots, k$ that point to nodes $v_0, \ldots, v_k$, respectively. A subset of nodes are labeled as corresponding to program variables.

An example input graph is displayed as "Input" in Fig. 3. In it, the node id (i.e., memory address) is displayed in the node. Edge labels correspond to specific fields in the program, e.g., 0 in our example corresponds to the `next` pointer in our example function from the previous section. For binary trees there are two more types of pointers `left` and `right` pointing to the left and right children of a tree node.

Output Representation Our aim is to mathematically describe the shape of the heap. In our model, we restrict ourselves to a syntactically restricted version of separation logic, in which formulas are of the form $\exists x_1, \ldots, x_n . a_1 * \ldots * a_m$, where each atomic formula a_i is either $\mathsf{ls}(x, y)$ (a list from x to y), $\mathsf{tree}(x)$ (a binary tree starting in x), or $\mathsf{none}(x)$ (no data structure at x). Existential quantifiers are used to give names to heap nodes which are needed to describe a shape, but not labeled by a program variable. For example, to describe a "panhandle list" (a list that ends in a cycle), the first list element on the cycle needs to be named. In separation logic, this can be expressed as $\exists t. \mathsf{ls}(x, t) * \mathsf{ls}(t, t)$.

Data We can generate synthetic (labeled) datasets for this problem. For this, we fix a set of predicates such as ls and tree (extensions could consider doubly-linked list segments, multi-trees, . . .) together with their inductive definitions. Then we enumerate separation logic formulas instantiating our predicates using a given set of program variables. Finally, for each formula, we enumerate heap graphs satisfying that formula. The result is a dataset consisting of pairs of heap graphs and associated formulas that are used by our learning procedures.

6.2 FORMULATION AS GGS-NNS

It is easy to obtain the node annotations for the intermediate prediction steps from the data generation process. So we train a variant of GGS-NN with observed annotations (observed at training time; not test time) to infer formulas from heap graphs. Note that it is also possible to use an unobserved GGS-

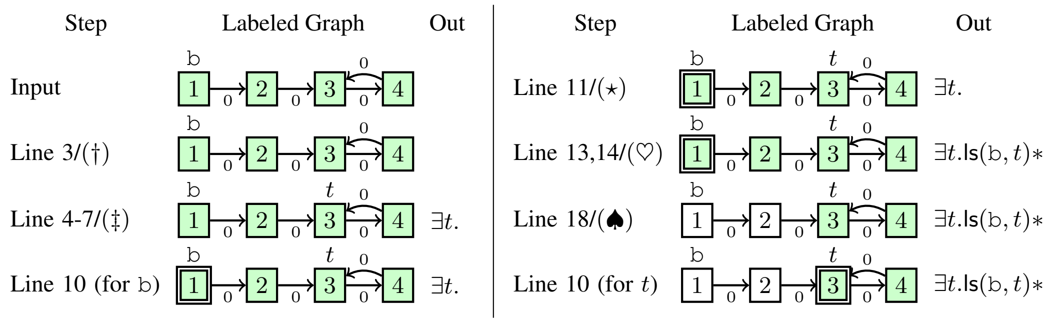


Figure 3: Illustration of the first 8 steps to predict a separation logic formula from a memory state. Label *is-named* signified by variable near node, *active* by double border, *is-explained* by white fill.

NN variant and do end-to-end learning. The procedure breaks down the production of a separation logic formula into a sequence of steps. We first decide whether to declare existential variables, and if so, choose which node corresponds to the variable. Once we have declared existentials, we iterate over all variable names and produce a separation logic formula describing the data structure rooted at the node corresponding to the current variable.

The full algorithm for predicting separation logic formula appears below, as Alg. 1. We use three explicit node annotations, namely *is-named* (heap node labeled by program variable or declared existentially quantified variable), *active* (cf. algorithm) and *is-explained* (heap node is part of data structure already predicted). Initial node labels can be directly computed from the input graph: "is-named" is on for nodes labeled by program variables, "active" and "is-explained" are always off (done in line 2). The commented lines in the algorithm are implemented using a GG-NN, i.e., Alg. 1 is an instance of our GGS-NN model. An illustration of the beginning of a run of the algorithm is shown in Fig. 3, where each step is related to one line of the algorithm.

Algorithm 1 Separation logic formula prediction procedure

```
Input: Heap graph $\mathcal{G}$ with named program variables
 1: $\boldsymbol{\mathcal{X}} \leftarrow$ compute initial labels from $\mathcal{G}$
 2: $\boldsymbol{\mathcal{H}} \leftarrow$ initialize node vectors by 0-extending $\boldsymbol{\mathcal{X}}$
 3: while $\exists$ quantifier needed do                          ▷ Graph-level Classification (†)
 4:     $t \leftarrow$ fresh variable name
 5:     $v \leftarrow$ pick node                                    ▷ Node Selection (‡)
 6:     $\boldsymbol{\mathcal{X}} \leftarrow$ turn on "is-named" for $v$ in $\boldsymbol{\mathcal{X}}$
 7:     print "$\exists t.$"
 8: end while
 9: for node $v_\ell$ with label "is-named" in $\boldsymbol{\mathcal{X}}$ do
10:     $\boldsymbol{\mathcal{H}} \leftarrow$ initialize node vectors, turn on "active" label for $v_\ell$ in $\boldsymbol{\mathcal{X}}$
11:     $pred \leftarrow$ pick data structure predicate           ▷ Graph-level Classification (⋆)
12:     if $pred = \mathsf{ls}$ then
13:         $\ell_{end} \leftarrow$ pick list end node              ▷ Node Selection (♡)
14:         print "$\mathsf{ls}(\ell, \ell_{end}) *$"
15:     else
16:         print "$pred(\ell) *$"
17:     end if
18:     $\boldsymbol{\mathcal{X}} \leftarrow$ update node annotations in $\boldsymbol{\mathcal{X}}$   ▷ Node Annotation (♠)
19: end for
```

6.3 Model Setup Details

We use the full GGS-NN model where $\mathcal{F}_{\boldsymbol{o}}^{(k)}$ and $\mathcal{F}_{\boldsymbol{\mathcal{X}}}^{(k)}$ have separate propagation models. For all the GG-NN components in the GGS-NN pipeline, we unrolled the propagation process for 10 time steps. The GGS-NNs associated with step (†) (deciding wheter more existentially quantified variable need to be declared) and (‡) (identify which node need to be declared as existentially quantified) uses $D = 16$ dimensional node representations. For all other GGS-NN components, $D = 8$ is used. Adam (Kingma & Ba, 2014) is used for optimization, the models are trained on minibatches of 20 graphs, and optimized until training error is very low. For the graph-level classification tasks, we also artificially balanced classes to have even number of examples from each class in each minibatch. All the GGS-NN components contain less than 5k parameters and no overfitting is observed during training.

6.4 Batch Prediction Details

In practice, a set of heap graphs will be given as input and a single output formula is expected to describe and be consistent with all the input graphs. The different heap graphs can be snapshots of the heap state at different points in the program execution process, or different runs of the same program with different inputs. We call this the "batch prediction" setup contrasting with the single graph prediction described in the main paper.

To make batch predictions, we run one GGS-NN for each graph simultaneously. For each prediction step, the outputs of all the GGS-NNs at that step across the batch of graphs are aggregated.

For node selection outputs, the common named variables link nodes on different graphs togeter, which is the key for aggregating predictions in a batch. We compute the score for a particular named variable t as $o_t = \sum_g o^g_{\mathcal{V}_g(t)}$, where $\mathcal{V}_g(t)$ maps variable name t to a node in graph g, and $o^g_{\mathcal{V}_g(t)}$ is the output score for named variable t in graph g. When applying a softmax over all names using o_t as scores, this is equivalent to a model that computes $p(toselect = t) = \prod_g p_g(toselect = \mathcal{V}_g(t))$.

For graph-level classification outputs, we add up scores of a particular class across the batch of graphs, or equivalently compute $p(class = k) = \prod_g p_g(class = k)$. Node annotation outputs are updated for each graph independently as different graphs have completely different set of nodes. However, when the algorithm tries to update the annotation for one named variable, the nodes associated with that variable in all graphs are updated. During training, all labels for intermediate steps are available to us from the data generation process, so the training process again can be decomposed to single output single graph training.

A more complex scenario allowing for nested data structures (e.g., list of lists) was discussed in Brockschmidt et al. (2015). We have also successfully extended the GGS-NN model to this case. More details on this can be found in Appendix C.

6.5 Experiments.

For this paper, we produced a dataset of 327 formulas that involves three program variables, with 498 graphs per formula, yielding around 160,000 formula/heap graph combinations. To evaluate, we split the data into training, validation and test sets using a 6:2:2 split on the formulas (i.e., the formulas in the test set were not in the training set). We measure correctness by whether the formula predicted at test time is logically equivalent to the ground truth; equivalence is approximated by canonicalizing names and order of the formulas and then comparing for exact equality.

We compared our GGS-NN-based model with a method we developed earlier (Brockschmidt et al., 2015). The earlier approach treats each prediction step as standard classification, and requires complex, manual, problem-specific feature engineering, to achieve an accuracy of 89.11%. In contrast, our new model was trained with no feature engineering and very little domain knowledge and achieved an accuracy of 89.96%.

An example heap graph and the corresponding separation logic formula found by our GGS-NN model is shown in Fig. 4. This example also involves nested data structures and the batching extension developed in the previous section.

We have also successfully used our new model in a program verification framework, supplying needed program invariants to a theorem prover to prove correctness of a collection of list-manipulating algorithms such as insertion sort. The following Table 4 lists a set of benchmark list manipulation programs and the separation logic formula invariants found by the GGS-NN model, which were successfully used in a verification framework to prove the correctness of corresponding programs.

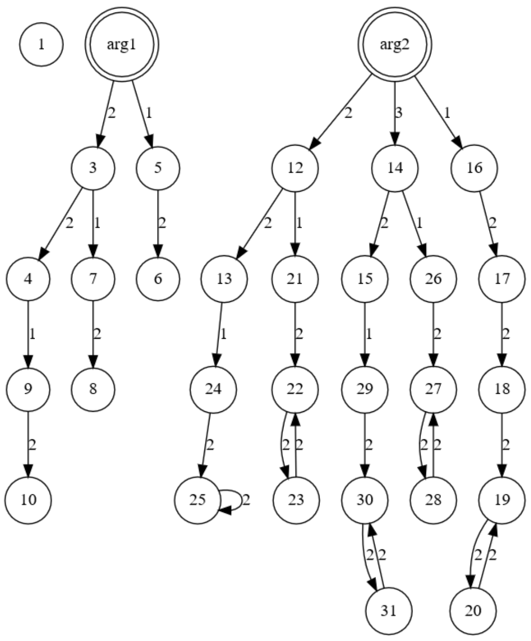


Figure 4: A heap graph example that contains two named variables `arg1` and `arg2`, and one isolated `NULL` node (node 1). All the edges to `NULL` are not shown here for clarity. The numbers on edges indicate different edge types. Our GGS-NN model successfully finds the right formula $\mathsf{ls}(\texttt{arg1}, \texttt{NULL}, \lambda t_1 \to \mathsf{ls}(t_1, \texttt{NULL}, \top)) * \mathsf{tree}(\texttt{arg2}, \lambda t_2 \to \exists e_1 . \mathsf{ls}(t_2, e_1, \top) * \mathsf{ls}(e_1, e_1, \top))$.

Program	Invariant Found
Traverse1	$\mathsf{ls}(\texttt{lst}, \texttt{curr}) * \mathsf{ls}(\texttt{curr}, \texttt{NULL})$
Traverse2	$\texttt{curr} \neq \texttt{NULL} * \texttt{lst} \neq \texttt{NULL} * \mathsf{ls}(\texttt{lst}, \texttt{curr}) * \mathsf{ls}(\texttt{curr}, \texttt{NULL})$
Concat	$\texttt{a} \neq \texttt{NULL} * a \neq b * b \neq \texttt{curr} * \texttt{curr} \neq \texttt{NULL}$ $* \mathsf{ls}(\texttt{curr}, \texttt{NULL}) * \mathsf{ls}(a, \texttt{curr}) * \mathsf{ls}(b, \texttt{NULL})$
Copy	$\mathsf{ls}(\texttt{curr}, \texttt{NULL}) * \mathsf{ls}(\texttt{lst}, \texttt{curr}) * \mathsf{ls}(\texttt{cp}, \texttt{NULL})$
Dispose	$\mathsf{ls}(\texttt{lst}, \texttt{NULL})$
Insert	$\texttt{curr} \neq \texttt{NULL} * \texttt{curr} \neq \texttt{elt} * \texttt{elt} \neq \texttt{NULL} * \texttt{elt} \neq \texttt{lst} * \texttt{lst} \neq \texttt{NULL}$ $* \mathsf{ls}(\texttt{elt}, \texttt{NULL}) * \mathsf{ls}(\texttt{lst}, \texttt{curr}) * \mathsf{ls}(\texttt{curr}, \texttt{NULL})$
Remove	$\texttt{curr} \neq \texttt{NULL} * \texttt{lst} \neq \texttt{NULL} * \mathsf{ls}(\texttt{lst}, \texttt{curr}) * \mathsf{ls}(\texttt{curr}, \texttt{NULL})$

Table 4: Example list manipulation programs and the separation logic formula invariants the GGS-NN model founds from a set of input graphs. The "$\neq$" parts are produced by a deterministic procedure that goes through all the named program variables in all graphs and checks for inequality.

A further extension of the current pipeline has been shown to be able to successfully prove more sophisticated programs like sorting programs and various other list-manipulating programs.

7 RELATED WORK

The most closely related work is GNNs, which we have discussed at length above. Micheli (2009) proposed another closely related model that differs from GNNs mainly in the output model. GNNs have been applied in several domains (Gori et al., 2005; Di Massa et al., 2006; Scarselli et al., 2009; Uwents et al., 2011), but they do not appear to be in widespread use in the ICLR community. Part of our aim here is to publicize GNNs as a useful and interesting neural network variant.

An analogy can be drawn between our adaptation from GNNs to GG-NNs, to the work of Domke (2011) and Stoyanov et al. (2011) in the structured prediction setting. There belief propagation (which must be run to near convergence to get good gradients) is replaced with truncated belief propagation updates, and then the model is trained so that the truncated iteration produce good results after a fixed number of iterations. Similarly, Recursive Neural Networks (Goller & Kuchler, 1996; Socher et al., 2011) being extended to Tree LSTMs (Tai et al., 2015) is analogous to our using of GRU updates in GG-NNs instead of the standard GNN recurrence with the aim of improving the long-term propagation of information across a graph structure.

The general idea expressed in this paper of assembling problem-specific neural networks as a composition of learned components has a long history, dating back at least to the work of Hinton (1988) on assembling neural networks according to a family tree structure in order to predict relations between people. Similar ideas appear in Hammer & Jain (2004) and Bottou (2014).

Graph kernels (Shervashidze et al., 2011; Kashima et al., 2003) can be used for a variety of kernel-based learning tasks with graph-structured inputs, but we are not aware of work that learns the kernels and outputs sequences. Perozzi et al. (2014) convert graphs into sequences by following random walks on the graph then learns node embeddings using sequence-based methods. Sperduti & Starita (1997) map graphs to graph vectors then classify using an output neural network. There are several models that make use of similar propagation of node representations on a graph structure. Bruna et al. (2013) generalize convolutions to graph structures. The difference between their work and GNNs is analogous to the difference between convolutional and recurrent networks. Duvenaud et al. (2015) also consider convolutional like operations on graphs, building a learnable, differentiable variant of a successful graph feature. Lusci et al. (2013) converts an arbitrary undirected graph to a number of different DAGs with different orientations and then propagates node representations inwards towards each root, training an ensemble of models. In all of the above, the focus is on one-step problems.

GNNs and our extensions have many of the same desirable properties of pointer networks (Vinyals et al., 2015); when using node selection output layers, nodes from the input can be chosen as outputs. There are two main differences: first, in GNNs the graph structure is explicit, which makes the models less general but may provide stronger generalization ability; second, pointer networks require that each node has properties (e.g., a location in space), while GNNs can represent nodes that are defined only by their position in the graph, which makes them more general along a different dimension.

GGS-NNs are related to soft alignment and attentional models (e.g., Bahdanau et al. (2014); Kumar et al. (2015); Sukhbaatar et al. (2015)) in two respects: first, the graph representation in Eq. 7 uses context to focus attention on which nodes are important to the current decision; second, node annotations in the program verification example keep track of which nodes have been explained so far, which gives an explicit mechanism for making sure that each node in the input has been used over the sequence of producing an output.

8 Discussion

What is being learned? It is instructive to consider what is being learned by the GG-NNs. To do so, we can draw analogy between how the bAbI task 15 would be solved via a logical formulation. As an example, consider the subset of lines needed to answer one example on the right.

```
B is E
E has_fear H
eval B has_fear
```

To do logical reasoning, we would need not only a logical encoding of the facts present in the story but also the background world knowledge encoded as inference rules such as

$$\texttt{is(x, y)} \wedge \texttt{has-fear(y, z)} \implies \texttt{has-fear(x, z)}. \tag{9}$$

Our encoding of the tasks simplifies the parsing of the story into graph form, but it does not provide any of the background knowledge. The GG-NN model can be seen as learning this, with results stored in the neural network weights.

Discussion The results in the paper show that GGS-NNs have desirable inductive biases across a range of problems that have some intrinsic graph structure to them, and we believe there to be many more cases where GGS-NNs will be useful. There are, however, some limitations that need to be overcome to make them apply even more broadly. Two limitations that we mentioned previously are that the bAbI task translation does not incorporate temporal order of inputs or ternary and higher order relations. We can imagine several possibilities for lifting these restrictions, such as concatenating a series of GG-NNs, where there is one GG-NNs for each edge, and representing higher order relations as factor graphs. A more significant challenge is how to handle less structured input representations. For example, in the bAbI tasks it would be desirable not to use the symbolic form of the inputs. One possible approach is to incorporate less structured inputs, and latent vectors, in our GGS-NNs. However, experimentation is needed to find the best way of addressing these issues.

The current GGS-NNs formulation specifies a question only after all the facts have been consumed. This implies that the network must try to derive all consequences of the seen facts and store all pertinent information to a node within its node representation. This is likely not ideal; it would be preferable to develop methods that take the question as an initial input, and then dynamically derive the facts needed to answer the question.

We are optimistic about the further applications of GGS-NNs. We are particularly interested in continuing to develop end-to-end learnable systems that can learn about semantic properties of programs, that can learn more complicated graph algorithms, and in applying these ideas to problems that require reasoning over knowledge bases and databases. More generally, we consider these graph neural networks as representing a step towards a model that can combine structured representations with the powerful algorithms of deep learning, with the aim of taking advantage of known structure while learning and inferring how to reason with and extend these representations.

ACKNOWLEDGEMENTS

We thank Siddharth Krishna, Alex Gaunt, Emine Yilmaz, Milad Shokouhi, and Pushmeet Kohli for useful conversations and Douwe Kiela for comments on an earlier draft of the paper.

REFERENCES

Almeida, Luis B. A learning rule for asynchronous perceptrons with feedback in a combinatorial environment. In *Artificial neural networks*, pp. 102–111. IEEE Press, 1990.

Bahdanau, Dzmitry, Cho, Kyunghyun, and Bengio, Yoshua. Neural machine translation by jointly learning to align and translate. *CoRR*, abs/1409.0473, 2014.

Bottou, Léon. From machine learning to machine reasoning. *Machine learning*, 94(2):133–149, 2014.

Brockschmidt, Marc, Chen, Yuxin, Cook, Byron, Kohli, Pushmeet, and Tarlow, Daniel. Learning to decipher the heap for program verification. In *Workshop on Constructive Machine Learning at the International Conference on Machine Learning (CMLICML)*, 2015.

Bruna, Joan, Zaremba, Wojciech, Szlam, Arthur, and LeCun, Yann. Spectral networks and locally connected networks on graphs. *arXiv preprint arXiv:1312.6203*, 2013.

Cho, Kyunghyun, Van Merriënboer, Bart, Gulcehre, Caglar, Bahdanau, Dzmitry, Bougares, Fethi, Schwenk, Holger, and Bengio, Yoshua. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

Di Massa, Vincenzo, Monfardini, Gabriele, Sarti, Lorenzo, Scarselli, Franco, Maggini, Marco, and Gori, Marco. A comparison between recursive neural networks and graph neural networks. In *International Joint Conference on Neural Networks (IJCNN)*, pp. 778–785. IEEE, 2006.

Domke, Justin. Parameter learning with truncated message-passing. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 2937–2943. IEEE, 2011.

Duvenaud, David, Maclaurin, Dougal, Aguilera-Iparraguirre, Jorge, Gómez-Bombarelli, Rafael, Hirzel, Timothy, Aspuru-Guzik, Alán, and Adams, Ryan P. Convolutional networks on graphs for learning molecular fingerprints. *arXiv preprint arXiv:1509.09292*, 2015.

Goller, Christoph and Kuchler, Andreas. Learning task-dependent distributed representations by backpropagation through structure. In *IEEE International Conference on Neural Networks*, volume 1, pp. 347–352. IEEE, 1996.

Gori, Marco, Monfardini, Gabriele, and Scarselli, Franco. A new model for learning in graph domains. In *International Joint Conference onNeural Networks (IJCNN)*, volume 2, pp. 729–734. IEEE, 2005.

Hammer, Barbara and Jain, Brijnesh J. Neural methods for non-standard data. In *European Symposium on Artificial Neural Networks (ESANN)*, 2004.

Hinton, Geoffrey E. Representing part-whole hierarchies in connectionist networks. In *Proceedings of the Tenth Annual Conference of the Cognitive Science Society*, pp. 48–54. Erlbaum., 1988.

Hoare, Charles Antony Richard. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

Kashima, Hisashi, Tsuda, Koji, and Inokuchi, Akihiro. Marginalized kernels between labeled graphs. In *Proceedings of the International Conference on Machine Learning*, volume 3, pp. 321–328, 2003.

Kingma, Diederik and Ba, Jimmy. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Kumar, Ankit, Irsoy, Ozan, Su, Jonathan, Bradbury, James, English, Robert, Pierce, Brian, Ondruska, Peter, Gulrajani, Ishaan, and Socher, Richard. Ask me anything: Dynamic memory networks for natural language processing. *arXiv preprint arXiv:1506.07285*, 2015.

Lusci, Alessandro, Pollastri, Gianluca, and Baldi, Pierre. Deep architectures and deep learning in chemoinformatics: the prediction of aqueous solubility for drug-like molecules. *J Chem Inf Model*, 2013.

Micheli, Alessio. Neural network for graphs: A contextual constructive approach. *IEEE Transactions on Neural Networks*, 20(3):498–511, 2009.

O'Hearn, Peter, Reynolds, John C., and Yang, Hongseok. Local reasoning about programs that alter data structures. In *15th International Workshop on Computer Science Logic (CSL'01)*, pp. 1–19, 2001.

Perozzi, Bryan, Al-Rfou, Rami, and Skiena, Steven. Deepwalk: Online learning of social representations. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pp. 701–710. ACM, 2014.

Pineda, Fernando J. Generalization of back-propagation to recurrent neural networks. *Physical review letters*, 59(19):2229, 1987.

Piskac, Ruzica, Wies, Thomas, and Zufferey, Damien. GRASShopper - complete heap verification with mixed specifications. In *20st International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'14)*, pp. 124–139, 2014.

Reynolds, John C. Separation logic: A logic for shared mutable data structures. In *7th IEEE Symposium on Logic in Computer Science (LICS'02)*, pp. 55–74, 2002.

Scarselli, Franco, Gori, Marco, Tsoi, Ah Chung, Hagenbuchner, Markus, and Monfardini, Gabriele. The graph neural network model. *IEEE Transactions on Neural Networks*, 20(1):61–80, 2009.

Shervashidze, Nino, Schweitzer, Pascal, Van Leeuwen, Erik Jan, Mehlhorn, Kurt, and Borgwardt, Karsten M. Weisfeiler-lehman graph kernels. *The Journal of Machine Learning Research*, 12: 2539–2561, 2011.

Socher, Richard, Lin, Cliff C, Manning, Chris, and Ng, Andrew Y. Parsing natural scenes and natural language with recursive neural networks. In *Proceedings of the 28th international conference on machine learning (ICML-11)*, pp. 129–136, 2011.

Sperduti, Alessandro and Starita, Antonina. Supervised neural networks for the classification of structures. *IEEE Transactions on Neural Networks*, 8(3):714–735, 1997.

Stoyanov, Veselin, Ropson, Alexander, and Eisner, Jason. Empirical risk minimization of graphical model parameters given approximate inference, decoding, and model structure. In *International Conference on Artificial Intelligence and Statistics*, pp. 725–733, 2011.

Sukhbaatar, Sainbayar, Szlam, Arthur, Weston, Jason, and Fergus, Rob. End-to-end memory networks. *arXiv preprint arXiv:1503.08895*, 2015.

Tai, Kai Sheng, Socher, Richard, and Manning, Christopher D. Improved semantic representations from tree-structured long short-term memory networks. *arXiv preprint arXiv:1503.00075*, 2015.

Uwents, Werner, Monfardini, Gabriele, Blockeel, Hendrik, Gori, Marco, and Scarselli, Franco. Neural networks for relational learning: an experimental comparison. *Machine Learning*, 82(3):315–349, 2011.

Vinyals, Oriol, Fortunato, Meire, and Jaitly, Navdeep. Pointer networks. *arXiv preprint arXiv:1506.03134*, 2015.

Weston, Jason, Bordes, Antoine, Chopra, Sumit, and Mikolov, Tomas. Towards ai-complete question answering: a set of prerequisite toy tasks. *arXiv preprint arXiv:1502.05698*, 2015.

A CONTRACTION MAP EXAMPLE

Consider a linear 1-hidden unit cycle-structured GNN with N nodes $\{1, \ldots, N\}$. For simplicity we ignored all edge labels and node labels, equivalently this is a simple example with $L_{\mathcal{V}} = 1$ and $L_{\mathcal{E}} = 1$. At each timestep t we update hidden states $h_1, \ldots, h_N$ as

$$h_i^{(t)} = m_i \cdot h_{i-1}^{(t-1)} + b_i, \tag{10}$$

for each i, where m_i and b_i are parameters of the propagation model. We use the convention that h_j cycles around and refers to h_{N+j} when $j \leq 0$. Let $\mathbf{h}^{(t)} = [h_1^{(t)}, \ldots, h_N^{(t)}]^\top$,

$$\mathbf{M} = \begin{bmatrix} 0 & 0 & 0 & \ldots & m_1 \\ m_2 & 0 & 0 & & 0 \\ 0 & m_3 & 0 & & 0 \\ & \vdots & & \ddots & \\ 0 & 0 & & m_N & 0 \end{bmatrix} \tag{11}$$

and $\mathbf{b} = [b_1, \ldots, b_N]^\top$. We can write the joint update for all i as

$$\mathbf{h}^{(t)} = \mathbf{M}\mathbf{h}^{(t-1)} + \mathbf{b} = T(\mathbf{h}^{(t-1)}) \tag{12}$$

Restrict the update to define a contraction mapping in the Euclidean metric. This means that there is some $\rho < 1$ such that for any $\mathbf{h}, \mathbf{h}'$,

$$||T(\mathbf{h}) - T(\mathbf{h}')|| < \rho||\mathbf{h} - \mathbf{h}'||, \tag{13}$$

or in other words,

$$||\mathbf{M}(\mathbf{h} - \mathbf{h}')|| < \rho||\mathbf{h} - \mathbf{h}'||. \tag{14}$$

We can immediately see that this implies that $|m_i| < \rho$ for each i by letting $\mathbf{h}$ be the elementary vector that is all zero except for a 1 in position $i - 1$ and letting $\mathbf{h}'$ be the all zeros vector.

Expanding Eq. 10, we get

$$\begin{aligned} h_i^{(t)} &= m_i \cdot (m_{i-1} h_{i-1}^{(t-2)} + b_{i-1}) + b_i \\ &= m_i m_{i-1} h_{i-1}^{(t-2)} + m_i b_{i-1} + b_i \\ &= m_i m_{i-1} (m_{i-2} h_{i-2}^{(t-3)} + b_{i-2}) + m_i b_{i-1} + b_i \\ &= m_i m_{i-1} m_{i-2} h_{i-2}^{(t-3)} + m_i m_{i-1} b_{i-2} + m_i b_{i-1} + b_i. \end{aligned} \tag{15}$$

In the GNN model, node label l_i controls which values of m_i and b_i are used during the propagation. Looking at this expansion and noting that $m_i < \rho$ for all i, we see that information about labels of nodes δ away will decay at a rate of $\left(\frac{1}{\rho}\right)^\delta$. Thus, at least in this simple case, the restriction that T be a contraction means that it is not able to maintain long-range dependencies.

A.1 NONLINEAR CASE

The same analysis can be applied to a nonlinear update, i.e.

$$h_i^{(t)} = \sigma\left(m_i \cdot h_{i-1}^{(t-1)} + b_i\right), \tag{16}$$

where σ is any nonlinear function. Then $T(\mathbf{h}) = \sigma(\mathbf{M}\mathbf{h} + \mathbf{b})$. Let $T(\mathbf{h}) = [T_1(\mathbf{h}), ..., T_N(\mathbf{h})]^\top$, where $T_i(\mathbf{h}^{(t-1)}) = h_i^{(t)}$. The contraction map definition Eq. 13 implies that each entry of the Jacobian matrix of T is bounded by ρ, i.e.

$$\left|\frac{\partial T_i}{\partial h_j}\right| < \rho, \qquad \forall i, \forall j. \tag{17}$$

To see this, consider two vectors $\mathbf{h}$ and $\mathbf{h}'$, where $h_k = h'_k, \forall k \neq j$ and $h_j + \Delta = h'_j$. The definition in Eq. 13 implies that for all i,

$$||T_i(\mathbf{h}) - T_i(\mathbf{h}')|| \leq ||T(\mathbf{h}) - T(\mathbf{h}')|| < \rho|\Delta|. \tag{18}$$

Therefore

$$\left\| \frac{T_i(h_1, ..., h_{j-1}, h_j, h_{j+1}, ..., h_N) - T_i(h_1, ..., h_{j-1}, h_j + \Delta, h_{j+1}, ..., h_N)}{\Delta} \right\| < \rho, \tag{19}$$

where the left hand side is $\left\| \frac{\partial T_i}{\partial h_j} \right\|$ by definition as $\Delta \to 0$.

When $j = i - 1$,

$$\left| \frac{\partial T_i}{\partial h_{i-1}} \right| < \rho. \tag{20}$$

Also, because of the special cycle graph structure, for all other js we have $\frac{\partial T_i}{\partial h_j} = 0$. Applying this to the update at timestep t, we get

$$\left| \frac{\partial h_i^{(t)}}{\partial h_{i-1}^{(t-1)}} \right| < \rho. \tag{21}$$

Now let's see how a change in $h_1^{(1)}$ could affect $h_t^{(t)}$. Using the chain rule and the special graph structure, we have

$$\begin{aligned} \left| \frac{\partial h_t^{(t)}}{\partial h_1^{(1)}} \right| &= \left| \frac{\partial h_t^{(t)}}{\partial h_{t-1}^{(t-1)}} \cdot \frac{\partial h_{t-1}^{(t-1)}}{\partial h_{t-2}^{(t-2)}} \cdots \frac{\partial h_2^{(2)}}{\partial h_1^{(1)}} \right| \\ &= \left| \frac{\partial h_t^{(t)}}{\partial h_{t-1}^{(t-1)}} \right| \cdot \left| \frac{\partial h_{t-1}^{(t-1)}}{\partial h_{t-2}^{(t-2)}} \right| \cdots \left| \frac{\partial h_2^{(2)}}{\partial h_1^{(1)}} \right| \\ &< \rho \cdot \rho \cdots \rho = \rho^{t-1}. \end{aligned} \tag{22}$$

As $\rho < 1$, this derivative will approach 0 exponentially fast as t grows. Intuitively, this means that the impact one node has on another node far away will decay exponetially, therefore making it difficult to model long range dependencies.

B WHY ARE RNN AND LSTM SO BAD ON THE SEQUENCE PREDICTION TASKS?

RNN and LSTM performance on the sequence prediction tasks, i.e. bAbI task 19, shortest path and Eulerian circuit, are very poor compared to single output tasks. The Eulerian circuit task is the one that RNN and LSTM fail most dramatically. A typical training example for this task looks like the following,

```
3 connected-to 7
7 connected-to 3
1 connected-to 2
2 connected-to 1
5 connected-to 7
7 connected-to 5
0 connected-to 4
4 connected-to 0
1 connected-to 0
0 connected-to 1
8 connected-to 6
6 connected-to 8
3 connected-to 6
6 connected-to 3
5 connected-to 8
8 connected-to 5
4 connected-to 2
2 connected-to 4
eval eulerian-circuit 5 7       5,7,3,6,8
```

This describes a graph with two cycles 3-7-5-8-6 and 1-2-4-0, where 3-7-5-8-6 is the target cycle and 1-2-4-0 is a smaller distractor graph. All edges are presented twice in both directions for symmetry. The task is to find the cycle that starts with the given two nodes and in the direction from the first to the second. The distractor graph is added to increase the difficulty of this task, this also makes the output cycle not strictly "Eulerian".

For RNN and LSTM the above training example is further transformed into a sequence of tokens,

```
n4 e1 n8 eol n8 e1 n4 eol n2 e1 n3 eol n3 e1 n2 eol n6 e1 n8 eol
n8 e1 n6 eol n1 e1 n5 eol n5 e1 n1 eol n2 e1 n1 eol n1 e1 n2 eol
n9 e1 n7 eol n7 e1 n9 eol n4 e1 n7 eol n7 e1 n4 eol n6 e1 n9 eol
n9 e1 n6 eol n5 e1 n3 eol n3 e1 n5 eol q1 n6 n8 ans 6 8 4 7 9
```

Note the node IDs here are different from the ones in the original symbolic data. The RNN and LSTM read through the whole sequence, and start to predict the first output when reading the `ans` token. Then for each prediction step, the `ans` token is fed as the input and the target node ID (treated as a class label) is expected as the output. In this current setup, the output of each prediction step is not fed as the input for the next. Our GGS-NN model uses the same setup, where the output of one step is not used as input to the next, only the predicted node annotations $\boldsymbol{\mathcal{X}}^{(k)}$ carry over from one step to the next, so the comparison is still fair for RNN and LSTM. Changing both our method and the baselines to make use of previous predictions is left as future work.

From this example we can see that the sequences the RNN and LSTM have to handle is quite long, close to 80 tokens before the predictions are made. Some predictions really depend on long range memory, for example the first edge (3-7) and first a few tokens (`n4 e1 n8`) in the sequence are needed to make prediction in the third prediction step (3 in the original symbolic data, and 4 in the tokenized RNN data). Keeping long range memory in RNNs is challenging, LSTMs do better than RNNs but still can't completely solve the problem.

Another challenge about this task is the output sequence does not appear in the same order as in the input sequence. In fact, the data has no sequential nature at all, even when the edges are randomly permutated, the target output sequence should not change. The same applies for bAbI task 19 and the shortest path task. GGS-NNs are good at handling this type of "static" data, while RNN and LSTM are not. However future work is needed to determine how best to apply GGS-NNs to temporal sequential data which RNN and LSTM are good at. This is one limitation of the GGS-NNs model which we discussed in Section 8.

C Nested Prediction Details

Data structures like *list of lists* are nested data structures, in which the `val` pointer of each node in a data structure points to another data structure. Such data structures can be represented in separation logic by allowing predicates to be nested. For example, a list of lists can be represented as $\mathsf{ls}(x, y, \lambda t \to \mathsf{ls}(t, \texttt{NULL}))$, where $\lambda t \to \mathsf{ls}(t, \texttt{NULL})$ is a lambda expression and says that for each node in the list from x to y, its `val` pointer t satisfies $\mathsf{ls}(t, \texttt{NULL})$. So there is a list from x to y, where each node in that list points to another list. A simple list without nested structures can be represented as $\mathsf{ls}(x, y, \top)$ where $\top$ represents an empty predicate. Note that unlike the non-nested case where the `val` pointer always points to `NULL`, we have to consider the `val` pointers here in order to describe and handle nested data structures.

To make our GGS-NNs able to predict nested formulas, we adapt Alg. 1 to Alg. 2. Where an outer loop goes through each named variable once and generate a nested predicate with the node associated with that variable as the active node. The nested prediction procedure handles prediction similarly as in Alg. 1. Before calling the nested prediction procedure recursively, the node annotation update in line 32 not only annotates nodes in the current structure as "is-explained", but also annotates nodes linked to via the "val" pointer from all nodes in the current structure as "active". For the list of lists example, after predicting "$\mathsf{ls}(x, y,$", the annotation step annotates all nodes in the list from x to y as "is-explained" and all nodes the `val` pointer points to from the list as "active". This knowledge is not hard coded into the algorithm, the annotation model can learn this behavior from data.

Algorithm 2 Nested separation logic formula prediction procedure

```
1: procedure OuterLoop($\mathcal{G}$)                          ▷ Graph $\mathcal{G}$ with named program variables
2:     $\boldsymbol{\mathcal{X}} \leftarrow$ compute initial labels from $\mathcal{G}$
3:     for each variable name var do
4:         $v_\ell \leftarrow$ node associated with var
5:         turn on "active" bit for $v_\ell$ in $\boldsymbol{\mathcal{X}}$
6:         PredictNestedFormula($\mathcal{G}$, $\boldsymbol{\mathcal{X}}$, var)
7:     end for
8: end procedure
9:
10: procedure PredictNestedFormula($\mathcal{G}$, $\boldsymbol{\mathcal{X}}$, var)
11:     $\boldsymbol{\mathcal{H}} \leftarrow$ initialize node vectors by 0-extending $\boldsymbol{\mathcal{X}}$
12:     while $\exists$ quantifier needed do                    ▷ Graph-level Classification (†)
13:         $e \leftarrow$ fresh existentially quantified variable name
14:         $v \leftarrow$ pick node                               ▷ Node Selection (‡)
15:         $\boldsymbol{\mathcal{X}} \leftarrow$ turn on "is-named" for $v$ in $\boldsymbol{\mathcal{X}}$
16:         print "$\exists e.$"
17:     end while
18:     if var is a lambda variable name then
19:         print "$\lambda$ var."
20:     end if
21:     $pred \leftarrow$ pick data structure predicate           ▷ Graph-level Classification ($\star$)
22:     if $pred =$ ls then
23:         $\ell_{end} \leftarrow$ pick list end node                ▷ Node Selection ($\heartsuit$)
24:         var$_{end} \leftarrow$ get variable name associated with $\ell_{end}$
25:         print "ls(var, var$_{end}$,"
26:     else if $pred =$ tree then
27:         print "tree(var,"
28:     else
29:         print "none(var) $*$"
30:         return
31:     end if
32:     $\boldsymbol{\mathcal{X}} \leftarrow$ update node annotations in $\boldsymbol{\mathcal{X}}$        ▷ Node Annotation ($\spadesuit$)
33:     $t \leftarrow$ fresh lambda variable name
34:     PredictNestedFormula($\mathcal{G}$, $\boldsymbol{\mathcal{X}}$, $t$)     ▷ Recursively predict all nested formulas.
35:     print ") $*$"
36: end procedure
```