A. Modified attention mechanism in ResRGAT

The query and key vectors for each attention head h (of in total H heads), $\mathbf{q}_v^{(h)} \in \mathbb{R}^D$ for node v and $\mathbf{k}_e^{(h)} \in \mathbb{R}^D$ for an edge $e = (u, v, r)$ ending in v, respectively, are computed using head-specific linear transformations parameterized by $\mathbf{Q}^{(h)} \in \mathbb{R}^{D/H \times D}$ and $\mathbf{K}^{(h)} \in \mathbb{R}^{D/H \times 2D}$:

$$\mathbf{q}_v^{(h)} = \mathbf{Q}^{(h)} \mathbf{h}_v \ , \tag{6}$$

$$\mathbf{k}_e^{(h)} = \mathbf{K}^{(h)} [\mu(\mathbf{h}_u, r); \mathbf{a}_r] \ , \tag{7}$$

where $[\cdot;\cdot]$ denotes the (vertical) concatenation of two (column) vectors and $\mathbf{a}_r$ is the same edge type-specific parameter vector that is used in Eqs. 2-4. Note that Eq. 6 is slightly different from a standard attention mechanism, which would compute key vectors as $\mathbf{K}^{(h)} \mu(\mathbf{h}_u, r)$. We expect that including the edge type vector $\mathbf{a}_r$ directly in the keys will make it easier to learn to place attention.

The attention score $w_e^{(h)}$ for an edge e and head h is computed by a scaled dot product and then normalized to obtain $\alpha^{(h)}$ as usual:

$$w_e^{(h)} = \frac{\mathbf{q}_v^{(h)} \cdot \mathbf{k}_e^{(h)}}{\sqrt{D/H}} \tag{8}$$

and

$$\alpha_e^{(h)} = \frac{\exp(w_e^{(h)})}{\sum_{e^* \in \mathcal{E}(\cdot,v)} \exp(w_{e^*}^{(h)})} \ . \tag{9}$$

Unlike multi-head attention in transformers and previous work (Veličković et al., 2018; Busbridge et al., 2019), we do not use linear transformations to obtain the value vectors. Instead, the vector $\mu(\mathbf{h}_u, r) = \boldsymbol{\mu}_e = [\boldsymbol{\mu}_e^{(1)}, \ldots, \boldsymbol{\mu}_e^{(H)}]$ is split in H equally sized chunks and the h-th part $\boldsymbol{\mu}_e^{(h)} \in \mathbb{R}^{D/H}$ is taken as value vector for the h-th head. The attention scores $\alpha_e^{(h)}$ and value vectors $\boldsymbol{\mu}_e^{(h)}$ are then used to compute the neighbourhood aggregation vector $\overline{\mathbf{h}}_v^{(h)}$ for head h as a weighted sum. The full neighbourhood aggregation vector $\overline{\mathbf{h}}_v$ is then given as a concatenation over all heads h:

$$\overline{\mathbf{h}}_v^{(h)} = \sum_{e \in \mathcal{E}(\cdot,v)} \alpha_e^{(h)} \boldsymbol{\mu}_e^{(h)} \ , \tag{10}$$

$$\overline{\mathbf{h}}_v = [\overline{\mathbf{h}}_v^{(1)}; \ldots; \overline{\mathbf{h}}_v^{(H)}] \ . \tag{11}$$

This change reduces the number of trainable parameters, and improves breadth-wise backpropagation.

B. Symmetrically Gated Recurrent Unit

As elaborated above, depth-wise residual update functions and GRUs as used in the GGNN improve depth-wise backpropagation towards lower-level features of the same node. To also improve the breadth-wise backpropagation, we propose to use an adapted version of the GRU. We call the proposed update function, which is described by the following equations, a *Symmetrically Gated Recurrent Unit* (SGRU):

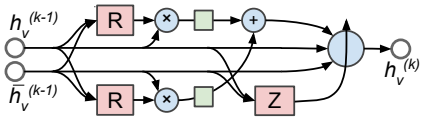

Figure 3. SGRU.

$$\mathbf{r}_h^{(k)} = \sigma(W_{r_h} \overline{\mathbf{h}}_v^{(k-1)} + U_{r_h} \mathbf{h}_v^{(k-1)} + b_{r_h}) \tag{12}$$

$$\mathbf{r}_x^{(k)} = \sigma(W_{r_x} \overline{\mathbf{h}}_v^{(k-1)} + U_{r_x} \mathbf{h}_v^{(k-1)} + b_{r_x}) \tag{13}$$

$$\mathbf{z}_x^{(k)} = W_{z_x} \overline{\mathbf{h}}_v^{(k-1)} + U_{z_x} \mathbf{h}_v^{(k-1)} + b_{z_x} \tag{14}$$

$$\mathbf{z}_h^{(k)} = W_{z_h} \overline{\mathbf{h}}_v^{(k-1)} + U_{z_h} \mathbf{h}_v^{(k-1)} + b_{z_h} \tag{15}$$

$$\mathbf{z}_u^{(k)} = W_{z_u} \overline{\mathbf{h}}_v^{(k-1)} + U_{z_u} \mathbf{h}_v^{(k-1)} + b_{z_u} \tag{16}$$

$$\hat{z}_{x,i}^{(k)}, \hat{z}_{h,i}^{(k)}, \hat{z}_{u,i}^{(k)} = \mathrm{softmax}([z_{x,i}^{(k)}, z_{h,i}^{(k)}, z_{u,i}^{(k)}]) \tag{17}$$

$$\hat{\mathbf{h}}_v^{(k)} = \tanh(W(\overline{\mathbf{h}}_v^{(k-1)} \odot \mathbf{r}_x^{(k)}) + U(\mathbf{h}_v^{(k-1)} \odot \mathbf{r}_h^{(k)})) \tag{18}$$

$$\mathbf{h}_v^{(k)} = \hat{\mathbf{z}}_x^{(k)} \odot \overline{\mathbf{h}}_v^{(k-1)} + \hat{\mathbf{z}}_h^{(k)} \odot \mathbf{h}_v^{(k-1)} + \hat{\mathbf{z}}_u^{(k)} \odot \hat{\mathbf{h}}_v^{(k)} \ . \tag{19}$$

The SGRU differs from the GRU used in the GGNN in (i) introducing an additional reset gate $\mathbf{r}_x^{(k)}$ that is applied to the aggregated neighbour states $\overline{\mathbf{h}}_v^{(k-1)}$ and (ii) computing the output state $\mathbf{h}_v^{(k)}$ as a three-way mixture between the previous node state $\mathbf{h}_v^{(k-1)}$, the aggregated neighbour states $\overline{\mathbf{h}}_v^{(k-1)}$, and the candidate state $\hat{\mathbf{h}}_v^{(k)}$ instead of a two-way mixture between $\mathbf{h}_v^{(k-1)}$ and $\hat{\mathbf{h}}_v^{(k)}$. The interpolation coefficients $\hat{\mathbf{z}}_x^{(k)}$, $\hat{\mathbf{z}}_h^{(k)}$, and $\hat{\mathbf{z}}_u^{(k)}$ are produced by an elementwise softmax between the three vectors $\mathbf{z}_x^{(k)}$, $\mathbf{z}_h^{(k)}$, and $\mathbf{z}_u^{(k)}$. See also Figure 3 for an illustration.

The vanilla GRU as applied to sequences has two inputs: the input vector x_t at a certain timestep t and the state vector h. While it implements additive gated updates on h, it applies linear transformations and nonlinearities on x_t before merging it into h. In the SGRU, both inputs are gated similarly, which lets the x_t input benefit from the same additive gradient behavior as h and gave rise to the name "symmetrically gated". This is crucial for improving horizontal information flow during the training of GNN since the x_t input of the SGRU receives the aggregated neighbourhood vector.

C. Baselines

C.1. A note on memory-efficiency of the baselines

When experimenting with the baselines (RGCN, GGNN, and RGAT), we quickly ran out of memory for larger hidden dimensions of the GNNs and larger numbers of layers. This was caused by the relation matrices, which have $O(n^2)$ parameters for every relation. In the implementations we used, the relations were indexed from a tensor and have to be retained in memory for every edge. While more memory-efficient implementations are possible, the existing ones sacrifice execution time, which can increase significantly with a large number of different relations.

Gradient accumulation significantly slows down training. As a simple fix to maintain reasonable training speed for our experiments, we replace $\mu_{\text{MM}}(\cdot)$ in our RGCN and GGNN with $\mu_{\text{MM-red}}(\mathbf{h}_u) = W_B W_r^* W_A \mathbf{h}_u$, where W_r^* is an edge type specific square matrix of lower dimensionality than W_r and W_A and W_B are matrices projecting into and out of W_r^*'s dimensionality that are shared for all edge types. Note that we did this only for the Tree Max experiments.

C.2. RGAT Baseline

Like in the work of Busbridge et al. (2019), we use relation-specific transformation matrices $\mathbf{W}^{(r)}$ and relation and head-specific query and key matrices $\mathbf{Q}^{(r,h)}$ and $\mathbf{K}^{(r,h)}$. First, we define relation-dependent representations for a node, which is computed based on its current state $\mathbf{z}^{(k-1)}$:

$$\mathbf{g}_i^{(r)} = \mathbf{W}^{(r)} \mathbf{z}_i^{(k-1)} \ . \tag{20}$$

Subsequently, for every head h, we define the relation-specific query, key and value projections as:

$$\mathbf{q}_i^{(h,r)} = \mathbf{Q}^{(h,r)} \mathbf{g}_i^{(r)} \tag{21}$$

$$\mathbf{k}_i^{(h,r)} = \mathbf{K}^{(h,r)} \mathbf{g}_i^{(r)} \tag{22}$$

$$\mathbf{v}_i^{(h,r)} = \mathbf{V}^{(h,r)} \mathbf{z}_i^{(k-1)} \tag{23}$$

We compute attention between messages $\boldsymbol{\mu}_x$, where x indexes over all edges in the input graph. First, we compute the attention score for a message as Vaswani et al. (2017)

$$s_{\boldsymbol{\mu}_x}^{(h)} = \mathbf{q}_i^{(h,r)} \cdot \mathbf{k}_j^{(h,r)} \ , \tag{24}$$

where $\boldsymbol{\mu}_x$ is the message sent along an edge $j \to i$ labeled by the relation r. Please note that a node i may receive multiple messages from a node j, and that those messages could contain the same relation.

The scores are normalized over all messages that node i receives:

$$\alpha_{\boldsymbol{\mu}_x}^{(h)} = \frac{s_{\boldsymbol{\mu}_x}^{(h)}}{\sum_{\boldsymbol{\mu}_{x'}} s_{\boldsymbol{\mu}_{x'}}^{(h)}} \ . \tag{25}$$

The normalized scores are used to compute a summary:

$$\mathbf{z}_i^{(h,k)} = \sum_{\boldsymbol{\mu}_x} \mathbf{v}_j^{(h,r)} \alpha_{\boldsymbol{\mu}_x}^{(h)} \ , \tag{26}$$

where r is the relation of $\boldsymbol{\mu}_x$ and j is the source node id and i is the target node id of message $\boldsymbol{\mu}_x$.

The updated representation for node i is then the concatenation over all heads, fed through the activation function σ:

$$\mathbf{z}_i^{(k)} = \sigma([\mathbf{z}_i^{(0,k)}, \ldots, \mathbf{z}_i^{(H,k)}]) \ . \tag{27}$$

We experiment with a ReLU, as well as with a linear σ.

D. Notes on relation-aware message functions

Many different message function implementations are possible that take into account the relation connecting a node to a neighbour. The default message function used in earlier literature is simply a matrix multiplication:

$$\mu_{\text{MM}}(\mathbf{h}_u, r) = \mathbf{W}_r \mathbf{h}_u \tag{28}$$

This can easily lead to overparameterization and leads to difficulties implementing the GNNs in a both memory-efficient and computationally efficient way.

In CompGCN, Vashishth et al. (2020) propose the following function:

$$\mu_{\text{CompGCN}}(\mathbf{h}_u, r) = \mathbf{W}_{\lambda(r)} \phi(\mathbf{h}_u, \mathbf{a}_r) \ , \tag{29}$$

where $\mathbf{W}_{\lambda(r)}$ is one of the three matrices: one for forward relations, one for reverse and one for self-edges, and $\phi(\mathbf{h}_u, \mathbf{a}_r)$ is a composition function that combines the neighbour state and relation vector. For the composition function, the authors explore functions inspired by knowledge graph embedding literature: subtraction ($\mathbf{h}_u - \mathbf{a}_r$), multiplication or circular-correlation.

While this provides an alternative solution to over-parameterizing relations, and enables more efficient implementation, it does not improve gradient behavior since the backpropagation paths to the lower node features as well as to distant nodes still contain linear transformation and non-linearities.

To improve backpropagation, we can make use of (gated or residual) skip connections. The residual version would have the following general form:

$$\mu_{\text{Res}}(\mathbf{h}_u, r) = \mu_{\text{X}}(\mathbf{h}_u, r) + \mathbf{h}_u \ , \tag{30}$$

where μ_{X} is some function. Similarly to transformers and ResNets, we choose to use a two-layer MLP, which is more expressive than μ_{MM}, and unlike the composition-based message functions in Vashishth et al. (2020), can effortlessly incorporate additional edge features that are not based on the relation embeddings.

D.1. A note on interchangeability of relations

Note that the simple choice of adding a relation vector, similarly to CompGCN's subtraction version, might pose problems since it loses information about which relation was associated with which neighbour:

$$(\mathbf{h}_u + \mathbf{a}_r) + (\mathbf{h}_w + \mathbf{a}_r) = (\mathbf{h}_w + \mathbf{a}_r) + (\mathbf{h}_u + \mathbf{a}_r) \ . \quad (31)$$

This is undesireable since it limits the expressive power of the network when using a sum aggregator. It is not clear to us how much this would affect models using attention-based aggregation, however, we prefer to avoid this since the attention-based aggregator may behave like a uniform sum, especially throughout the early stages of training.

This leads to the following condition that we impose on the message function: The message function μ must be chosen such that

$$\mu(\mathbf{a}, b) + \mu(\mathbf{c}, d) \neq \mu(\mathbf{a}, d) + \mu(\mathbf{c}, b) \quad (32)$$

iff $a \neq c$ and $b \neq d$. The gated message function μ_{GCM} proposed earlier in our work satisfies this condition.

E. Backpropagation to distant nodes in Gated Graph Neural Networks

The GGNN (Li et al., 2015) implements the update function $\phi(\cdot)$ based on GRUs:

$$\mathbf{r}^{(k)} = \sigma(W_r \overline{\mathbf{h}}_v^{(k-1)} + U_r \mathbf{h}_v^{(k-1)} + b_r) \ , \quad (33)$$

$$\mathbf{z}^{(k)} = \sigma(W_z \overline{\mathbf{h}}_v^{(k-1)} + U_z \mathbf{h}_v^{(k-1)} + b_z) \ , \quad (34)$$

$$\hat{\mathbf{h}}_v^{(k)} = \tanh(W \overline{\mathbf{h}}_v^{(k-1)} + U(\mathbf{h}_v^{(k-1)} \odot \mathbf{r}^{(k)})) \ , \quad (35)$$

$$\mathbf{h}_v^{(k)} = (1 - \mathbf{z}^{(k)}) \odot \mathbf{h}_v^{(k-1)} + \mathbf{z}^{(k)} \odot \hat{\mathbf{h}}_v^{(k)} \ , \quad (36)$$

where $\overline{\mathbf{h}}_v^{(k-1)} = \gamma(\{\mu(\mathbf{h}_u^{(k-1)}, r)\}_{(u,v,r) \in \mathcal{E}(\cdot,v)})$ is the vector representing the aggregated neighbourhood of node v.

Consider a node classification task, where the top-level state $\mathbf{h}_v^{(K)}$ of node v is fed into a classifier that produces a loss $\mathcal{L}_v$. To compute the gradient $\nabla_{\mathbf{w}} \mathcal{L}_v = \dfrac{\partial \mathcal{L}_v}{\partial \mathbf{w}}$, where $\mathbf{w}$ are the parameters of the model, the chain rule is applied as follows (where we only take into account node v's contribution to the gradient, and ignore the contribution of other nodes):

$$\nabla_{\mathbf{w}} \mathcal{L}_v = \frac{\partial \mathcal{L}_v}{\partial \mathbf{w}} = \sum_{k=0}^{K} \frac{\partial \mathcal{L}_v}{\partial \mathbf{h}_v^{(k)}} \frac{\partial \mathbf{h}_v^{(k)}}{\partial \mathbf{w}} \ , \quad (37)$$

with

$$\frac{\partial \mathcal{L}_v}{\partial \mathbf{h}_v^{(k)}} = \frac{\partial \mathcal{L}_v}{\partial \mathbf{h}_v^{(K)}} \prod_{j=k+1}^{K} \frac{\partial \mathbf{h}_v^{(j)}}{\partial \mathbf{h}_v^{(j-1)}} \ . \quad (38)$$

Given the gated update equations of the GRU, the partial derivatives $\dfrac{\partial \mathbf{h}_v^{(k)}}{\partial \mathbf{h}_v^{(k-1)}}$ of the updated node representation w.r.t. the previous node representation are:

$$\frac{\partial \mathbf{h}_v^{(k)}}{\partial \mathbf{h}_v^{(k-1)}} = (1 - \mathbf{z}) \odot \mathbf{I} + \mathbf{z} \odot \frac{\partial \hat{\mathbf{h}}_v^{(k)}}{\partial \mathbf{h}_v^{(k-1)}} + \frac{\partial (-\mathbf{z})}{\partial \mathbf{h}_v^{(k-1)}} \odot \mathbf{h}_v^{(k-1)} + \frac{\partial \mathbf{z}}{\partial \mathbf{h}_v^{(k-1)}} \odot \hat{\mathbf{h}}_v^{(k)} \ . \quad (39)$$

The closer $\mathbf{z}$ is to the zero vector, the closer the partial derivatives across layers $\dfrac{\partial \mathbf{h}_v^{(k)}}{\partial \mathbf{h}_v^{(k-1)}}$ get to the identity matrix $\mathbf{I}$ and the more accurate the following approximation gets:

$$\frac{\partial \mathcal{L}_v}{\partial \mathbf{h}_v^{(k)}} \approx \frac{\partial \mathcal{L}_v}{\partial \mathbf{h}_v^{(K)}} \ , \quad (40)$$

which has the effect that all the GGNN layers are skipped. In practice, the gate $\mathbf{z}$ will take on different values throughout training, but the additive update still allows to retain some contribution of the original gradient $\dfrac{\partial \mathcal{L}_v}{\partial \mathbf{h}_v^{(K)}}$ deeper into the network.

On the other hand, consider a (K-1)-layer GGNN applied to a graph representing a sequence of length K. The sequence is represented as a graph according to the same rules as described in Section 5.1: the nodes are connected using only `next`-edges. Thus, only left-to-right propagation in the graph can be performed, and every node has only one incoming message, except the first, which has none. This is the setup that maximizes the ratio of graph diameter per number of nodes and edges.

When a (K-1)-layer GGNN is applied to such a graph of K nodes, there is only one backpropagation path from node K to node 0: $\dfrac{\partial \mathbf{h}_K^{(K)}}{\partial \mathbf{h}_0^{(0)}}$, which is also the longest backpropagation path possible in this graph:

$$\frac{\partial \mathbf{h}_K^{(K)}}{\partial \mathbf{h}_0^{(0)}} = \prod_{k=1}^{K} \frac{\partial \mathbf{h}_k^{(k)}}{\partial \mathbf{h}_{k-1}^{(k-1)}} \quad (41)$$

5 Ignoring the used message function μ_{MM} and considering every node has only one neighbour, the neighbourhood aggregation vector is simply the neighbour node vector: $\overline{\mathbf{h}}_v^{(k-1)} = \mathbf{h}_{k-1}^{(k-1)}$.

Plugging this into the GGNN equations (Eqs. 33-36) gives:

$$\begin{aligned}\frac{\partial \mathbf{h}_k^{(k)}}{\partial \mathbf{h}_{k-1}^{(k-1)}} &= \frac{\partial \mathbf{z}^{(k)} \tanh(W\mathbf{h}_{k-1}^{(k-1)})}{\partial \mathbf{h}_{k-1}^{(k-1)}} \\ &= \tanh(W\mathbf{h}_{k-1}^{(k-1)}) \frac{\partial \mathbf{z}^{(k)}}{\partial \mathbf{h}_{k-1}^{(k-1)}} \\ &+ \mathbf{z}^{(k)} \frac{\partial \tanh(W\mathbf{h}_{k-1}^{(k-1)})}{\partial \mathbf{h}_{k-1}^{(k-1)}} \\ &= \tanh(W\mathbf{h}_{k-1}^{(k-1)}) \frac{\partial \sigma(W_z\mathbf{h}_{k-1}^{(k-1)})}{\partial \mathbf{h}_{k-1}^{(k-1)}} \\ &+ \mathbf{z}^{(k)} \frac{\partial \tanh(W\mathbf{h}_{k-1}^{(k-1)})}{\partial \mathbf{h}_{k-1}^{(k-1)}} \end{aligned} \tag{42}$$

Both terms contain a partial derivative of the form $\frac{\partial \sigma(W\mathbf{h}_{k-1}^{(k-1)})}{\partial \mathbf{h}_{k-1}^{(k-1)}}$, which is essentially identical to the terms found in the backpropagation equations for a vanilla RNN.

As in vanilla RNNs (without any gating like in GRUs and LSTMs), these terms can cause vanishing gradients due to repeated multiplication with the derivative of a σ or $\tanh$ nonlinearity and vanishing or exploding gradients depending on the values of W (which also change throughout training and thus could potentially reach unstable values).

In addition to the gradient-based argument outlined above, residual networks (as well as GRUs and LSTMs), benefit from better generalization abilities due to effects which were not formalized or deeply studied in the literature known to us. Intuitively, the argument for residual networks (He et al., 2016) is that the individual residual blocks can easily learn to *not* contribute if necessary, which enables to learn very deep architectures, and otherwise, they only need to model the function that models the *residual* error computed from the output of the previous layer. In addition, we believe that adding such skip connections enables more effective weight sharing across layers and nodes.

F. Conditional Recall Experimental Details

In our experiments for this task, we randomly explore hyperparameter values from the following ranges for the compared models: (1) dimensionality of node feature vectors in $[100, 150, 200, 300]$ (larger values can cause memory issues with the baselines), (2) dropout in $[0., 0.1, 0.25, 0.5]$, (3) we usually set the dropout of embedding vectors to 0.1 (4) learning rate in $[0.001, 0.0005, 0.0001]$. We train for at most 200 epochs and use early stopping using validation accuracy. Throughout all of the experiments, we use the Adam (Kingma & Ba, 2015) optimizer. We also use label smoothing with a factor 0.1 everywhere. For our experiments with task length 15, we usually use the best hyperparameters found for length 10 for all methods and also experiment with smaller dropout values.

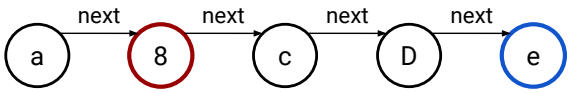

Figure 4. Example of input graph for the Conditional Recall task. The top-level state of the blue node is used for prediction. The red node specifies the desired output.

For the final reported accuracies, we trained with the best hyperparameters and loaded the model with the best validation error and evaluated on the test set. This was repeated with three different seeds that were shared between the tested methods.

The initial node states are initialized by embedding the node type using a low-dimensional embedding matrix (dimensionality 20; for a vocabulary of 62 characters) and projecting the low-dimensional embeddings to the node state dimension using a learned transformation.

An example of a graph used in this task is given in Figure 4. The output in this example should be "8".

G. Tree Max Example

See Figure 5 on page 16.

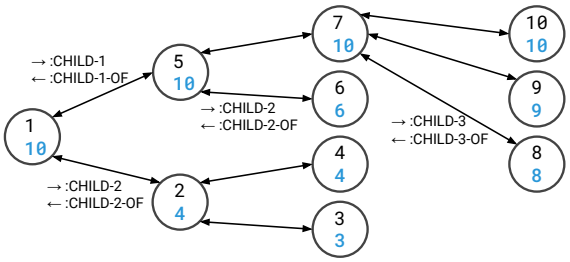

Figure 5. Example of input graph for the Tree Max task. The double-ended arrow between a node v and a node u represents two edges: one going from v to u and the other from u to v. Not every arrow is labeled for a clearer presentation. The labels on the arrows indicate edge labels: the forward arrow $\rightarrow$ corresponds to the edge label for going from the parent to the child and $\leftarrow$ for the reverse. The input labels are the black numbers and the output labels are the blue numbers. Note that we use a semi-supervised version of this task, where we erase the output labels of a large portion of the nodes, i.e. ignore their output labels in training and testing.

H. Tree Max Experimental Details:

During data generation, we first randomly pick a tree depth between five and fifteen. Then, we generate a tree, choosing between 0, 2 or 3 children for each node until we reach the chosen tree depth. The largest trees in our generated data contained over 200 nodes. The generated dataset contained 800 examples; we used a 50/25/25 train/validation/test split.

Not every node in the dataset is annotated. For every example, which consists of one graph with a maximum depth of N, the label of all nodes which are at most $N/2$ away from their "answer" node (i.e. the node which contains their output label) are ignored (i.e. not used in training or testing). Of the remaining nodes, which require to propagate information along a paths containing at least $N/2$ edges to estimate their label, foronly 50% the label is provided for training and testing. This setup defines a semi-supervised node classification task. We use it instead a fully supervised version providing all node labels, since the latter can be solved iteratively using immediate neighbour states only. This allows all GNNs to reach high node classification accuracy. In the semi-supervised version, the models have to learn what information to retain in unlabeled nodes, which are in between labeled nodes and their corresponding "answer" nodes.

For every model tested, we perform a hyperparameter search using all predefined seeds. For all models, we experiment only with dimensionality of 150 to avoid memory issues and ensure a fair comparison regarding representation dimensionality. Then we run the best hyperparameter setting with three different random seeds and report the test results in Table 2. Note that the same seeds are re-used for experiments for all models, and that every seed results in a different dataset being generated. We used early stopping and reloaded the best model based on element-wise accuracy on the validation set. Patience was set to 10 epochs but every experiment was run for a minimum of 50 epochs.

The models were evaluated using node-wise and graph-wise accuracy. The graph-level accuracy is 100% for an example only if all gold-labeled nodes in the graph have been classified correctly, and is 0% otherwise.

In our experiments, we randomly explore different combinations of different hyperparameter settings: dropout is selected from $\{0., 0.1, 0.25, 0.5\}$, dimensionality of node vectors is fixed to 150 for all methods. Learning rate is chosen from $\{0.001, 0.000333, 0.0001\}$. For all baselines, the number of layers was chosen from $10, 17$. Note however that 10 layers is insufficient to perfectly solve the task.

I. Ablation without residual connection in the message function.

See Table 5 for the results on the conditional recall task for the ablation where the residual connection in the edge-wise message function is removed.

Table 5. Conditional recall results. Average accuracy on the test set is reported. N is sequence length. L is the number of layers.

	N=5	N=10	N=15	N=5 (L=16)
ResRGAT, no. res. msg.	98.6 ± 1.4	95.1 ± 0.7	84.7 ± 6.7	88.4 ± 5.3