Pointer Graph Networks

Petar Veličković [1] **Lars Buesing** [1] **Matthew C. Overlan** [1] **Razvan Pascanu** [1] **Oriol Vinyals** [1] **Charles Blundell** [1]

Abstract

Graph neural networks (GNNs) are typically applied to static graphs that are assumed to be known upfront. This static input structure is often informed purely by insight of the machine learning practitioner, and might not be optimal for the actual task the GNN is solving. We introduce Pointer Graph Networks (PGNs) which augment sets or graphs with additional inferred edges for improved model expressivity. PGNs allow each node to dynamically *point* to another node, followed by message passing over these pointers. Despite its sparsity, this adaptable graph structure proves sufficiently expressive to simulate complex algorithms. The pointing mechanism is supervised to model long-term sequences of operations on classical data structures. PGNs can learn parallelisable variants of pointer-based data structures, and generalise out-of-distribution to 5× larger test inputs on dynamic graph connectivity tasks, outperforming unrestricted GNNs and Deep Sets.

1. Introduction

For problems where a graph structure is known (or can be approximated), graph neural networks (GNNs) often thrive. This places a burden upon the practitioner: which graph structure should be used? As the complexity of the task imposed on the GNN increases and, separately, the number of nodes increases, not allowing the choice of graph structure to be data-driven limits the applicability of GNNs.

Classical algorithms (Cormen et al., 2009) span computations that can be substantially more expressive than typical machine learning subroutines (e.g. matrix multiplications), making them hard to learn, and a good benchmark for GNNs (Chen et al., 2020; Dwivedi et al., 2020). Prior work has explored learning primitive algorithms (e.g. arithmetic) by RNNs (Zaremba & Sutskever, 2014; Kaiser & Sutskever, 2015; Trask et al., 2018), neural approximations to NP-hard problems (Vinyals et al., 2015; Kool et al., 2018), making GNNs learn (and transfer between) graph algorithms (Veličković et al., 2019; Georgiev & Lió, 2020), recently recovering a single neural core (Yan et al., 2020) capable of sorting, path-finding and binary addition. Here, we propose **Pointer Graph Networks** (PGNs), a framework that further *expands* the space of general-purpose algorithms that can be neurally executed. We consider algorithms that do not directly align to dynamic programming, relying upon pointer-based data structures.

Our PGNs adaptively learn a linear number of *pointer* edges between nodes. The pointers are optimised through *direct supervision* on classical data structures. We empirically demonstrate that PGNs further increase GNN expressivity beyond those with static graph structures (Garg et al., 2020), without sacrificing computational cost or sparsity for this added flexibility in graph structure.

2. Problem setup and PGN architecture

Problem setup We consider the following sequential supervised learning setting: Assume an underlying set of n entities. Given are sequences of inputs $\mathcal{E}^{(1)}, \mathcal{E}^{(2)}, \ldots$ where $\mathcal{E}^{(t)} = (\vec{e}_1^{(t)}, \vec{e}_2^{(t)}, \ldots, \vec{e}_n^{(t)})$ is defined by a list of feature vectors $\vec{e}_i^{(t)} \in \mathbb{R}^m$ for every entity $i \in \{1, \ldots, n\}$. We will suggestively refer to $\vec{e}_i^{(t)}$ as an *operation* on entity i at time t. The task consists now in predicting target outputs $\vec{y}^{(t)} \in \mathbb{R}^l$ based on operation sequence $\mathcal{E}^{(1)}, \ldots, \mathcal{E}^{(t)}$ up to t.

Pointer Graph Networks As the above sequential prediction task is defined on the underlying, un-ordered set of entities, any generalising prediction model is required to be *invariant under permutations* of the entity set. Furthermore, successfully predicting target $\vec{y}^{(t)}$ in general requires the prediction model to maintain a robust *data structure* to represent the history of operations for all entities throughout their lifetime. In the following we present our proposed prediction model, the Pointer Graph Network (PGN), that combines these desiderata in an efficient way.

At each step t, our PGN model computes *latent features* $\vec{h}_i^{(t)} \in \mathbb{R}^k$ for each entity i. Initially, $\vec{h}_i^{(0)} = \vec{0}$. Further, the PGN model determines dynamic *pointers*—one per en-

[1]DeepMind. Correspondence to: Petar Veličković <petarv@google.com>.

Proceedings of the 37th International Conference on Machine Learning, Vienna, Austria, PMLR 119, 2020. Copyright 2020 by the author(s).

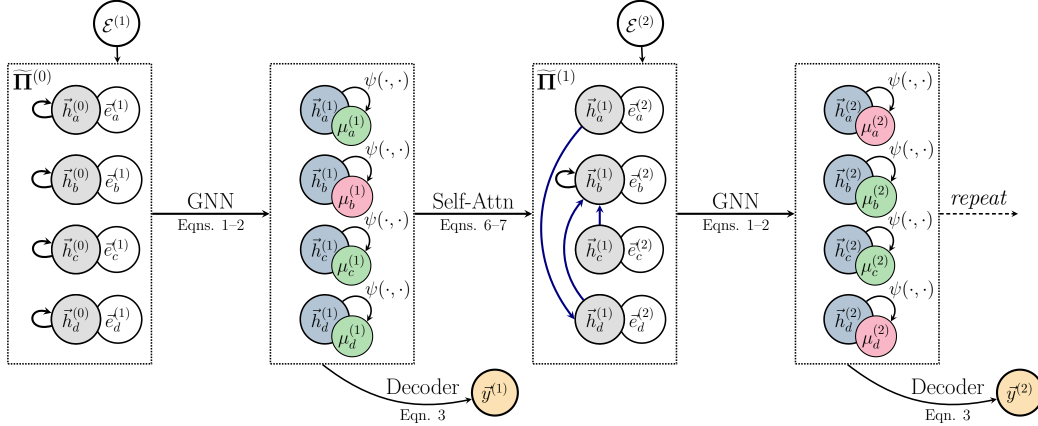

Figure 1. High-level overview of the pointer graph network (PGN) dataflow. Using descriptions of entity operations ($\vec{e}_i^{(t)}$), PGNs re-estimate latents $\vec{h}_i^{(t)}$, masks $\mu_i^{(t)}$, and (asymmetric) pointers $\widetilde{\mathbf{\Pi}}^{(t)}$. The *symmetrised* pointers, $\mathbf{\Pi}^{(t)}$, are then used as edges for a GNN that computes next-step latents, $\vec{h}_i^{(t+1)}$, continuing the process. The latents may be used to provide answers, $\vec{y}^{(t)}$, to queries about the underlying data. We highlight masked out nodes in **red**, and modified pointers/latents in **blue**.

tity and time step—which may be summarised in a pointer adjacency matrix $\mathbf{\Pi}^{(t)} \in \mathbb{R}^{n\times n}$. Initially, we assume each node points to itself: $\mathbf{\Pi}^{(0)} = \mathbf{I}_n$. A summary of the coming discussion may be found in Figure 1.

The Pointer Graph Network follows closely the *encode-process-decode* (Hamrick et al., 2018) paradigm. The current operation is encoded together with the latents in each entity using an *encoder* network f:

$$\vec{z}_i^{(t)} = f\left(\vec{e}_i^{(t)}, \vec{h}_i^{(t-1)}\right) \tag{1}$$

after which the derived entity representations $\mathbf{Z}^{(t)} = (\vec{z}_1^{(t)}, \vec{z}_2^{(t)}, \dots, \vec{z}_n^{(t)})$ are given to a *processor network*, P, which takes into account the current pointer adjacency matrix as relational information:

$$\mathbf{H}^{(t)} = P\left(\mathbf{Z}^{(t)}, \mathbf{\Pi}^{(t-1)}\right) \tag{2}$$

yielding next-step latent features, $\mathbf{H}^{(t)} = (\vec{h}_1^{(t)}, \vec{h}_2^{(t)}, \dots, \vec{h}_n^{(t)})$. These latents can be used to answer set-level queries using a *decoder* network g:

$$\vec{y}^{(t)} = g\left(\bigoplus_i \vec{z}_i^{(t)}, \bigoplus_i \vec{h}_i^{(t)}\right) \tag{3}$$

where $\bigoplus$ is any permutation-invariant *readout* aggregator, such as summation or maximisation.

Many efficient data structures only modify a small[1] subset of the entities at once (Cormen et al., 2009). We can incorporate this inductive bias into PGNs by masking their pointer modifications through a sparse *mask* $\mu_i^{(t)} \in \{0, 1\}$ for each node that is generated by a *masking* network ψ:

$$\mathbb{P}\left(\mu_i^{(t)} = 1\right) = \psi\left(\vec{z}_i^{(t)}, \vec{h}_i^{(t)}\right) \tag{4}$$

where the output activation function for ψ is the logistic sigmoid function, enforcing the probabilistic interpretation. In practice, we threshold the output of ψ as follows:

$$\mu_i^{(t)} = \mathbb{I}_{\psi\left(\vec{z}_i^{(t)}, \vec{h}_i^{(t)}\right) > 0.5} \tag{5}$$

The PGN now re-estimates the pointer adjacency matrix $\mathbf{\Pi}^{(t)}$ using $\vec{h}_i^{(t)}$. To do this, we leverage self-attention (Vaswani et al., 2017), computing all-pairs dot products between queries and keys:

$$\alpha_{ij}^{(t)} = \mathrm{softmax}_j\left(\left\langle \mathbf{W}_q \vec{h}_i^{(t)}, \mathbf{W}_k \vec{h}_j^{(t)} \right\rangle\right) \tag{6}$$

where $\mathbf{W}_q$ and $\mathbf{W}_k$ are learnable linear transformations, and $\langle\cdot,\cdot\rangle$ is the dot product operator. $\alpha_{ij}^{(t)}$ indicates the relevance of entity j to entity i, and we derive the pointer for i by choosing entity j with the *maximal* α_{ij}. To simplify the dataflow, we found it beneficial to *symmetrise* this matrix:

$$\widetilde{\mathbf{\Pi}}_{ij}^{(t)} = \mu_i^{(t)} \widetilde{\mathbf{\Pi}}_{ij}^{(t-1)} + \left(1 - \mu_i^{(t)}\right) \mathbb{I}_{j = \mathrm{argmax}_k \left(\alpha_{ik}^{(t)}\right)} \tag{7}$$

$$\mathbf{\Pi}_{ij}^{(t)} = \widetilde{\mathbf{\Pi}}_{ij}^{(t)} \vee \widetilde{\mathbf{\Pi}}_{ji}^{(t)} \tag{8}$$

where $\mathbb{I}$ is the indicator function, $\widetilde{\mathbf{\Pi}}^{(t)}$ denotes the pointers *before symmetrisation*, and $\vee$ denotes logical disjunction between the two operands. Nodes i and j will be linked together in $\mathbf{\Pi}^{(t)}$ if j is the most relevant to i, or vice-versa.

Unlike prior work which relied on the Gumbel trick (Kazi et al., 2020; Kipf et al., 2018), we will provide direct supervision w.r.t. *ground-truth* pointers, $\hat{\mathbf{\Pi}}^{(t)}$, of a target data structure. Further, our data-driven *conditional masking* is reminiscent of neural execution engines (Yan et al., 2020).

[1] Typically on the order of $O(\log n)$ elements.

PGN components and optimisation In our implementation, the encoder, decoder, masking and key/query networks are all linear layers. Echoing the results of prior work on algorithmic modelling with GNNs (Veličković et al., 2019), we recovered strongest performance when using *message passing neural networks* (MPNNs) (Gilmer et al., 2017) for P, with elementwise maximisation aggregator. Hence, the computation of Equation 2 is realised as follows:

$$\vec{h}_i^{(t)} = U\left(\vec{z}_i^{(t)}, \max_{\mathbf{\Pi}_{ji}^{(t-1)}=1} M\left(\vec{z}_i^{(t)}, \vec{z}_j^{(t)}\right)\right) \tag{9}$$

where M and U are linear layers producing *vector messages*, followed by ReLU. Accordingly, we found that elementwise-max was the best readout operation for $\bigoplus$ in Equation 3.

Besides the query loss in $\vec{y}^{(t)}$ (Equation 3), PGNs optimise cross-entropy of the attentional coefficients $\alpha_{ij}^{(t)}$ (Equation 6) against the ground-truth pointers, $\hat{\mathbf{\Pi}}^{(t)}$, and binary cross-entropy of the masking network ψ (Equation 4) against the ground-truth entities being modified at time t.

3. Task: Dynamic graph connectivity

We focus on instances of the *dynamic graph connectivity* setup to illustrate the benefits of PGNs. Dynamic connectivity querying is an important subroutine in computer science, e.g. when computing *minimum spanning trees*—deciding if an edge can be included without inducing cycles (Kruskal, 1956), or *maximum flow*—detecting existence of paths from source to sink with available capacity (Dinic, 1970).

Consider undirected and unweighted graphs of n nodes, with evolving edge sets through time; we denote the edges at time t by $E^{(t)}$. Initially, assume the graphs to be disconnected: $E^{(0)} = \emptyset$. At each step, an edge (u, v) may be added to or removed from $E^{(t-1)}$, yielding $E^{(t)} = E^{(t-1)} \ominus \{(u, v)\}$, where $\ominus$ is symmetric difference.

A *connectivity* query is then defined as follows: for a given pair of vertices (u, v), does there exist a path between them using edges in $E^{(t)}$? This yields binary ground-truth query answers $\hat{y}^{(t)}$ which we can supervise towards.

Incremental graph connectivity with *disjoint-set unions* We initially consider *incremental* graph connectivity: edges can only be *added* to the graph. Therefore, this problem only requires maintaining *disjoint sets*, supporting an efficient `union(u, v)` operation that performs a union of the sets containing u and v. Querying connectivity then simply requires checking whether u and v are in the same set, requiring an efficient `find(u)` operation which will retrieve the set containing u.

We consider a combined operation on (u, v): first, query whether u and v are connected, then perform a union on them if they are not. Pseudocode for this `query-union(u, v)` operation is given in Figure 2.

```
INIT(u)
1  π̂_u = u
2  r_u ~ U(0,1)

FIND(u)
1  if π̂_u ≠ u
2      π̂_u = FIND(π̂_u)
3  return π̂_u

UNION(u, v)
1  x = FIND(u)
2  y = FIND(v)
3  if x ≠ y
4      if r_x < r_y
5          π̂_x = y
6      else π̂_y = x

QUERY-UNION(u, v)
1  if FIND(u) = FIND(v)
2      return 0 // ŷ^(t) = 0
3  else UNION(u, v)
4  return 1 // ŷ^(t) = 1
```

Figure 2. Pseudocode of DSU operations. Manipulations of ground-truth pointers $\hat{\mathbf{\Pi}}$ ($\hat{\pi}_u$ for node u) are in blue; *path compression* heuristic highlighted in green.

We imitate the tree-based *disjoint-set union* (DSU) data structure (Galler & Fisher, 1964), known to yield optimal complexity (Fredman & Saks, 1989) for this task. DSU represents sets as *rooted trees*—each node, u, has a *parent pointer*, $\hat{\pi}_u$—and the set identifier will be its *root node*, ρ_u, which by convention points to itself ($\hat{\pi}_{\rho_u} = \rho_u$). Hence, `find(u)` reduces to recursively calling `find(pi[u])` until the root is reached—see Figure 2 (Left). Further, *path compression* (Tarjan & Van Leeuwen, 1984) is applied: upon calling `find(u)`, all nodes on the path from u to ρ_u will point to ρ_u. This self-organisation substantially reduces future querying time along the path.

`union(u, v)` reduces to finding u and v's roots, then making one of them point to the other. To avoid pointer ambiguity, we assign a random *priority*, $r_u \sim \mathcal{U}(0, 1)$, to every node at initialisation time, then always preferring the node with higher priority as the new root[2].

At each step t, we call `query-union(u, v)`, specified by operation descriptions $\vec{e}_i^{(t)} = r_i \| \mathbb{I}_{i=u \vee i=v}$, containing the nodes' priorities, along with a binary feature indicating which nodes are u and v. The corresponding output $\hat{y}^{(t)}$ indicates the return value of `query-union(u, v)`. We supervise the PGN's (asymmetric) pointers, $\widetilde{\mathbf{\Pi}}^{(t)}$, by making them match the ground-truth DSU's pointers, $\hat{\pi}_i$ ($\hat{\mathbf{\Pi}}_{ij}^{(t)} = 1$ iff $\hat{\pi}_i = j$, zero otherwise). Ground-truth masks, $\hat{\mu}_i^{(t)}$, are set to 0 for only the paths from u and v to their respective roots—no other node's state is changed.

Fully dynamic tree connectivity with *link/cut trees* We also consider *fully dynamic tree* connectivity—edges may now be removed, with the restriction that $E^{(t)}$ is acyclic. This is solvable in amortised $O(\log n)$ time by *link/cut trees* (LCTs) (Sleator & Tarjan, 1983), elegant data structures that maintain forests of *rooted trees*, requiring only one pointer per node. For brevity, we delegate further descriptions of their operations to Appendix C.

[2]Random priorities result in a sub-optimal worst-case time complexity, but one which is still decidedly sub-linear.

Table 1. F_1 scores on the dynamic graph connectivity tasks for all models considered, on five random seeds. All models are trained on $n = 20, \text{ops} = 30$ and tested on larger test sets.

Model	Disjoint-set union			Link/cut tree		
	$n = 20$ ops $= 30$	$n = 50$ ops $= 75$	$n = 100$ ops $= 150$	$n = 20$ ops $= 30$	$n = 50$ ops $= 75$	$n = 100$ ops $= 150$
GNN	$0.892_{\pm.007}$	$0.851_{\pm.048}$	$0.733_{\pm.114}$	$0.558_{\pm.044}$	$0.510_{\pm.079}$	$0.401_{\pm.123}$
Deep Sets	$0.870_{\pm.029}$	$0.720_{\pm.132}$	$0.547_{\pm.217}$	$0.515_{\pm.080}$	$0.488_{\pm.074}$	$0.441_{\pm.068}$
PGN-NM	$\mathbf{0.910}_{\pm.011}$	$0.628_{\pm.071}$	$0.499_{\pm.096}$	$0.524_{\pm.063}$	$0.367_{\pm.018}$	$0.353_{\pm.029}$
PGN	$0.895_{\pm.006}$	$\mathbf{0.887}_{\pm.008}$	$\mathbf{0.866}_{\pm.011}$	$\mathbf{0.651}_{\pm.017}$	$\mathbf{0.624}_{\pm.016}$	$\mathbf{0.616}_{\pm.009}$
PGN-Ptrs	$0.902_{\pm.010}$	$0.902_{\pm.008}$	$0.889_{\pm.007}$	$0.630_{\pm.022}$	$0.603_{\pm.036}$	$0.546_{\pm.110}$
Oracle-Ptrs	$0.944_{\pm.006}$	$0.964_{\pm.007}$	$0.968_{\pm.013}$	$0.776_{\pm.011}$	$0.744_{\pm.026}$	$0.636_{\pm.065}$

4. Evaluation and results

Experimental setup As in (Veličković et al., 2019; Yan et al., 2020), we evaluate *out-of-distribution generalisation*—training on random operation sequences for small input sets ($n = 20$ entities with $\text{ops} = 30$ operations), then testing on up to $5\times$ larger inputs ($n = 50, \text{ops} = 75$ and $n = 100, \text{ops} = 150$). In line with (Veličković et al., 2019), we generate 70 sequences for training, and 35 sequences across each test size category for testing.

All models compute $k = 32$ latent features in each layer, and are trained for $5,000$ epochs using Adam (Kingma & Ba, 2014) with learning rate of 0.005.

We evaluate the **PGN** model against three baselines:

Deep Sets (Zaheer et al., 2017) independently process individual entities, followed by an aggregation layer for resolving queries. This yields an only-self-pointer mechanism, $\mathbf{\Pi}^{(t)} = \mathbf{I}_n$ for all t, within our framework. Deep Sets are popular for set-based summary statistic tasks.

(Unrestricted) GNNs (Gilmer et al., 2017; Santoro et al., 2017; Xu et al., 2019) make no prior assumptions on node connectivity, yielding an all-ones adjacency matrix: $\mathbf{\Pi}_{ij}^{(t)} = 1$ for all (t, i, j). Such models are a popular choice when relational structure is assumed but not known.

PGN *without masks* (PGN-NM) remove the masking mechanism of Equations 4–7. This repeatedly overwrites all pointers, i.e. $\mu_i^{(t)} = 0$ for all (i, t). PGN-NM is related to a directly-supervised variant of the prior art in learnable k-NN graphs (Franceschi et al., 2019; Kazi et al., 2020).

To assess the utility of the data structure learnt by the PGN mechanism, as well as its performance limits, we perform two tests with *fixed pointers*, supervised only on the query: **PGN-Ptrs**, where pointers are fixed based on a pre-trained PGN model; and **Oracle-Ptrs**, which hard-codes the ground-truth pointers $\hat{\mathbf{\Pi}}^{(t)}$.

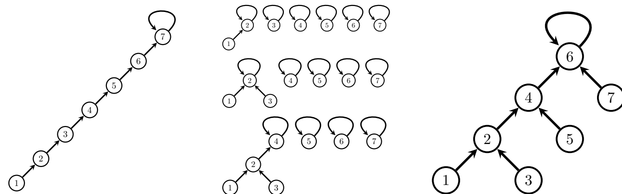

Figure 3. PGN rollout on the DSU setup, for the case of repeated `union(i, i+1)` (**Left**). The first few pointers in $\mathbf{\Pi}^{(t)}$ are visualised (**Middle**) as well as the final state (**Right**)—the PGN produced a valid DSU at all times, but $2\times$ shallower.

Results and discussion Our results (Table 1) indicate outperformance of our PGN model, especially on the larger-scale test sets. Competitive performance of PGN-Ptrs implies that the PGN models a robust data structure that GNNs can readily reuse. While the PGN-NM model is potent *in-distribution*, its performance rapidly decays once it is tasked to model larger sets of pointers at test time. We provide qualitative insights into PGNs' performance in Appendix D.

Rollout analysis of PGN pointers We determined that PGNs have 50–75% pointer accuracy w.r.t. ground-truth pointers $\hat{\mathbf{\Pi}}$, indicating indicates a substantial *deviation*, while maintaining strong query performance. The learnt pointers are still *meaningful*: given our sparse inductive bias, even minor discrepancies that result in modelling invalid data structures can negatively affect performance.

We observe the learnt PGN pointers on a pathological DSU example (Figure 3). Repeatedly calling `query-union(i, i+1)` with nodes ordered by priority yields a *linearised* DSU. Such graphs (of large diameter) are difficult for message propagation with GNNs. During rollout, the PGN models a correct DSU at all times, but *halving* its depth—easing GNN usage and GPU parallelisability. Effectively, the PGN learns to use the query supervision from $y^{(t)}$ to "nudge" its pointers in a direction more amenable to GNNs, discovering *parallelisable* data structures which may deviate from the ground-truth $\hat{\mathbf{\Pi}}^{(t)}$.

References

Chen, Z., Chen, L., Villar, S., and Bruna, J. Can graph neural networks count substructures? *arXiv preprint arXiv:2002.04025*, 2020.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. *Introduction to algorithms*. MIT press, 2009.

Dinic, E. A. Algorithm for solution of a problem of maximum flow in networks with power estimation. In *Soviet Math. Doklady*, volume 11, pp. 1277–1280, 1970.

Dwivedi, V. P., Joshi, C. K., Laurent, T., Bengio, Y., and Bresson, X. Benchmarking graph neural networks. *arXiv preprint arXiv:2003.00982*, 2020.

Franceschi, L., Niepert, M., Pontil, M., and He, X. Learning discrete structures for graph neural networks. *arXiv preprint arXiv:1903.11960*, 2019.

Fredman, M. and Saks, M. The cell probe complexity of dynamic data structures. In *Proceedings of the twenty-first annual ACM symposium on Theory of computing*, pp. 345–354, 1989.

Galler, B. A. and Fisher, M. J. An improved equivalence algorithm. *Communications of the ACM*, 7(5):301–303, 1964.

Garg, V. K., Jegelka, S., and Jaakkola, T. Generalization and representational limits of graph neural networks. *arXiv preprint arXiv:2002.06157*, 2020.

Georgiev, D. and Lió, P. Neural bipartite matching. *arXiv preprint arXiv:2005.11304*, 2020.

Gilmer, J., Schoenholz, S. S., Riley, P. F., Vinyals, O., and Dahl, G. E. Neural message passing for quantum chemistry. *arXiv preprint arXiv:1704.01212*, 2017.

Hamrick, J. B., Allen, K. R., Bapst, V., Zhu, T., McKee, K. R., Tenenbaum, J. B., and Battaglia, P. W. Relational inductive bias for physical construction in humans and machines. *arXiv preprint arXiv:1806.01203*, 2018.

Kaiser, Ł. and Sutskever, I. Neural gpus learn algorithms. *arXiv preprint arXiv:1511.08228*, 2015.

Kazi, A., Cosmo, L., Navab, N., and Bronstein, M. Differentiable graph module (dgm) graph convolutional networks. *arXiv preprint arXiv:2002.04999*, 2020.

Kingma, D. P. and Ba, J. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

Kipf, T., Fetaya, E., Wang, K.-C., Welling, M., and Zemel, R. Neural relational inference for interacting systems. *arXiv preprint arXiv:1802.04687*, 2018.

Kool, W., van Hoof, H., and Welling, M. Attention, learn to solve routing problems! *arXiv preprint arXiv:1803.08475*, 2018.

Kruskal, J. B. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical society*, 7(1):48–50, 1956.

Santoro, A., Raposo, D., Barrett, D. G., Malinowski, M., Pascanu, R., Battaglia, P., and Lillicrap, T. A simple neural network module for relational reasoning. In *Advances in neural information processing systems*, pp. 4967–4976, 2017.

Sleator, D. D. and Tarjan, R. E. A data structure for dynamic trees. *Journal of computer and system sciences*, 26(3): 362–391, 1983.

Tarjan, R. E. and Van Leeuwen, J. Worst-case analysis of set union algorithms. *Journal of the ACM (JACM)*, 31(2): 245–281, 1984.

Tarjan, R. E. and Werneck, R. F. Dynamic trees in practice. *Journal of Experimental Algorithmics (JEA)*, 14: 4–5, 2010.

Trask, A., Hill, F., Reed, S. E., Rae, J., Dyer, C., and Blunsom, P. Neural arithmetic logic units. In *Advances in Neural Information Processing Systems*, pp. 8035–8044, 2018.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. Attention is all you need. In *Advances in neural information processing systems*, pp. 5998–6008, 2017.

Veličković, P., Ying, R., Padovano, M., Hadsell, R., and Blundell, C. Neural execution of graph algorithms. *arXiv preprint arXiv:1910.10593*, 2019.

Vinyals, O., Fortunato, M., and Jaitly, N. Pointer networks. In *Advances in Neural Information Processing Systems*, pp. 2692–2700, 2015.

Xu, K., Li, J., Zhang, M., Du, S. S., Kawarabayashi, K.-i., and Jegelka, S. What can neural networks reason about? *arXiv preprint arXiv:1905.13211*, 2019.

Yan, Y., Swersky, K., Koutra, D., Ranganathan, P., and Hashemi, M. Neural execution engines. 2020. URL https://openreview.net/forum?id=rJg7BA4YDr.

Zaheer, M., Kottur, S., Ravanbakhsh, S., Poczos, B., Salakhutdinov, R. R., and Smola, A. J. Deep sets. In *Advances in neural information processing systems*, pp. 3391–3401, 2017.

Zaremba, W. and Sutskever, I. Learning to execute. *arXiv preprint arXiv:1410.4615*, 2014.

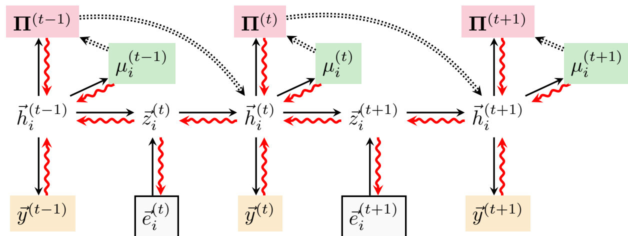

Figure 4. Detailed view of the dataflow within the PGN model, highlighting inputs $\vec{e}_i^{(t)}$ (*outlined*), objects optimised against ground-truths (query answers $\vec{y}^{(t)}$, masks $\mu_i^{(t)}$ and pointers $\boldsymbol{\Pi}^{(t)}$) (*shaded*) and all intermediate latent states ($\vec{z}_i^{(t)}$ and $\vec{h}_i^{(t)}$). Solid lines indicate differentiable computation with gradient flow in **red**, while dotted lines indicate non-differentiable opeations (*teacher-forced* at training time). **N.B.** This computation graph should also include edges from $\vec{z}_i^{(t)}$ into the query answers, masks and pointers (as it gets concatenated with $\vec{h}_i^{(t)}$)—we omit these edges for clarity.

A. Pointer graph networks gradient computation

To provide a more high-level overview of the PGN model's dataflow across all relevant variables (and for realising its computational graph and differentiability), we provide the visualisation in Figure 4.

Most operations of the PGN are realised as standard neural network layers and are hence differentiable; the two exceptions are the thresholding operations that decide the final masks $\mu_i^{(t)}$ and pointers $\boldsymbol{\Pi}^{(t)}$, based on the soft coefficients computed by the masking network and the self-attention, respectively. This makes no difference to the training algorithm, as the masks and pointers are *teacher-forced*, and the soft coefficients are directly optimised against ground-truth values of $\hat{\mu}_i^{(t)}$ and $\hat{\boldsymbol{\Pi}}^{(t)}$.

Further, note that our setup allows a clear path to end-to-end backpropagation (through the latent vectors) at all steps, allowing the representation of $\vec{h}_i^{(t)}$ to be optimised with respect to *all* predictions made for steps $t' > t$ in the future.

B. Summary of operation descriptions and supervision signals

To aid clarity, within Table 2, we provide an overview of all the operation descriptions and outputs (supervision signals) for the data structures considered here (disjoint-set unions and link/cut trees).

Note that the manipulation of ground-truth pointers ($\hat{\pi}_i$) is not discussed for LCTs in the main text for purposes of brevity; for more details, consult Appendix C.

C. Link/cut tree operations

In this section, we provide a detailed overview of the link/cut tree (LCT) data structure (Sleator & Tarjan, 1983), as well as the various operations it supports. This appendix is designed to be as self-contained as possible.

The operations supported by LCTs are: `find-root(u)` retrieves the root of node u; `link(u, v)` links nodes u and v, with the precondition that u is the root of its own tree; `cut(v)` removes the edge from v to its parent; `evert(u)` re-roots u's tree, such that u becomes the new root.

LCTs also support efficient *path-aggregate* queries on the (unique) path from u to v, which is very useful for reasoning on dynamic trees. Originally, this speeded up bottleneck computations in network flow algorithms (Dinic, 1970). Nowadays, the LCT has found usage across online versions of many classical graph algorithms, such as *minimum spanning forests* and *shortest paths* (Tarjan & Werneck, 2010). Here, however, we focus only on checking connectivity of u and v; hence `find-root(u)` will be sufficient for our queries.

Similarly to our DSU analysis, here we will compress updates and queries into one operation, `query-toggle(u, v)`, which our models will attempt to support. This operation first calls `evert(u)`, then checks if u and v are connected: if they are not, adding the edge between them wouldn't introduce cycles (and u is now the root of its tree), so `link(u, v)` is performed. Otherwise, `cut(v)` is performed—it is guaranteed to succeed, as v is not going to be the root node. Pseudocode of `query-toggle(u, v)`, along with visualising the effects of running it, is provided in Figure 5.

We encode each `query-toggle(u, v)` as $\vec{e}_i^{(t)} = r_i \| \mathbb{I}_{i=u \vee i=v}$. Random priorities, r_i, are again used; this time to determine whether u or v will be the node to call `evert` on, breaking ambiguity. As for DSU, we supervise the asymmetric pointers, $\widetilde{\boldsymbol{\Pi}}^{(t)}$, using the ground-truth LCT's pointers, $\hat{\pi}_i$ and ground-truth mask values, $\hat{\mu}_i^{(t)}$, are set to 0 only if $\hat{\pi}_i$ is modified in the operation at time t.

Before covering the specifics of LCT operations, it is important to understand how it *represents* the forest it models; namely, in order to support efficient $O(\log n)$ operations and path queries, the pointers used by the LCT can differ significantly from the edges in the forest being modelled.

Preferred path decomposition Many design choices in LCTs follow the principle of *"most-recent access"*: if a node was recently accessed, it is likely to get accessed again soon—hence we should keep it in a location that makes it easily accessible.

Table 2. Summary of operation descriptions and supervision signals on the data structures considered.

Data structure and operation	**Operation descriptions, $\vec{e}_i^{(t)}$**	**Supervision signals**
Disjoint-set union (Galler & Fisher, 1964) `query-union(u, v)`	r_i: randomly sampled priority of node i, $\mathbb{I}_{i=u \vee i=v}$: Is node i being operated on?	$\hat{y}^{(t)}$: are u and v in the same set?, $\hat{\mu}_i^{(t)}$: is node i visited by `find(u)` or `find(v)`?, $\hat{\mathbf{\Pi}}_{ij}^{(t)}$: is $\hat{\pi}_i = j$ after executing? (*asymmetric* pointer)
Link/cut tree (Sleator & Tarjan, 1983) `query-toggle(u, v)`	r_i: randomly sampled priority of node i, $\mathbb{I}_{i=u \vee i=v}$: Is node i being operated on?	$\hat{y}^{(t)}$: are u and v connected?, $\hat{\mu}_i^{(t)}$: is node i visited during `query-toggle(u, v)`?, $\hat{\mathbf{\Pi}}_{ij}^{(t)}$: is $\hat{\pi}_i = j$ after executing? (*asymmetric* pointer)

QUERY-TOGGLE(u, v)

1. **if** $r_u < r_v$
2. SWAP(u, v)
3. EVERT(u)
4. **if** FIND-ROOT$(v) \neq u$
5. LINK(u, v)
6. **return** 0 **//** $\hat{y}^{(t)} = 0$
7. **else** CUT(v)
8. **return** 1 **//** $\hat{y}^{(t)} = 1$

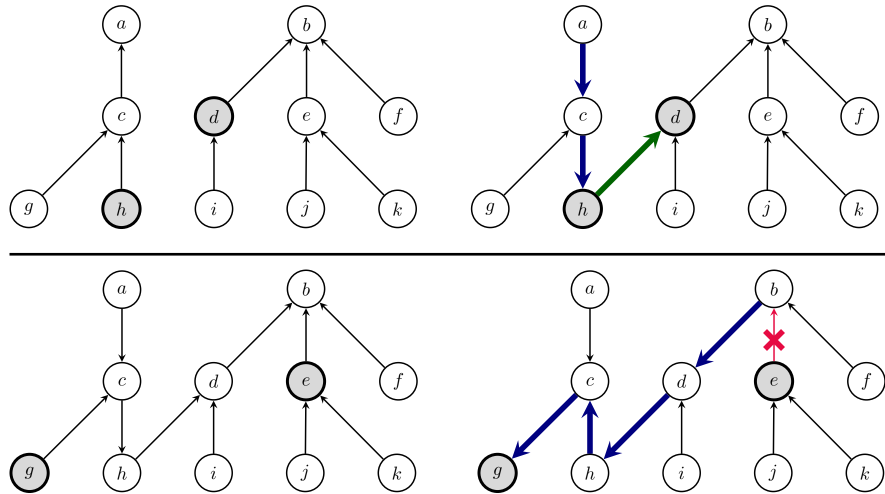

Figure 5. **Left:** Pseudocode of the `query-toggle(u, v)` operation, which will be handled by our models; **Right:** Effect of calling `query-toggle(h, d)` on a specific forest (**Top**), followed by calling `query-toggle(g, e)` (**Bottom**). Edges affected by `evert` (**blue**), `link` (**green**), and `cut` (**red**) are highlighted. **N.B.** this figure represents changes to the forest being modelled, and *not* the underlying LCT pointers; see Appendix C for more information on pointer manipulation.

The first such design is *preferred path decomposition*: the modelled forest is partitioned into **preferred paths**, such that each node may have at most one *preferred child*: the child most-recently accessed during a node-to-root operation. As we will see soon, *any* LCT operation on a node u will involve looking up the path to its respective root ρ_u—hence every LCT operation will be be composed of several node-to-root operations.

One example of a preferred path decomposition is demonstrated in Figure 6 (Left). Note how each node may have *at most one* preferred child. When a node is not a preferred child, its parent edge is used to *jump between paths*, and is hence often called a **path-parent**.

LCT pointers Each preferred path is represented by LCTs in a way that enables fast access—in a binary search tree (BST) keyed by depth. This implies that the nodes along the path will be stored in a binary tree (each node will potentially have a *left* and/or *right* child) which respects the following recursive invariant: for each node, all nodes in its left subtree will be *closer* to the root, and all nodes in its right subtree will be *further* from the root.

For now, it is sufficient to recall the invariant above—the specific implementation of binary search trees used in LCTs will be discussed towards the end of this section. It should be apparent that these trees should be *balanced*: for each node, its left and right subtree should be of (roughly) comparable sizes, recovering an optimal lookup complexity of $O(\log n)$, for a BST of n nodes.

Each of the preferred-path BSTs will specify its own set of pointers. Additionally, we still need to include the *path-parents*, to allow recombining information across different preferred paths. While we could keep these links unchanged, it is in fact canonical to place the path-parent in the **root** node of the path's BST (**N.B.** this node may be different from the top-of-path node[3]!).

As we will notice, this will enable more elegant operation of the LCT, and further ensures that each LCT node will have *exactly one parent pointer* (either in-BST parent or path-parent, allowing for jumping between different path BSTs), which aligns perfectly with our PGN model assumptions.

The ground-truth pointers of LCTs, $\hat{\mathbf{\Pi}}^{(t)}$, are then recovered as all the parent pointers contained within these binary search trees, along with all the path-parents. Similarly, ground-truth masks, $\hat{\mu}_i^{(t)}$, will be the subset of LCT nodes whose pointers may change during the operation at time t. We illustrate how a preferred path decomposition can be represented with LCTs within Figure 6 (Right).

[3]The top-of-path node is always the *minimum* node of the BST, obtained by recursively following left-children, starting from the root node, while possible.

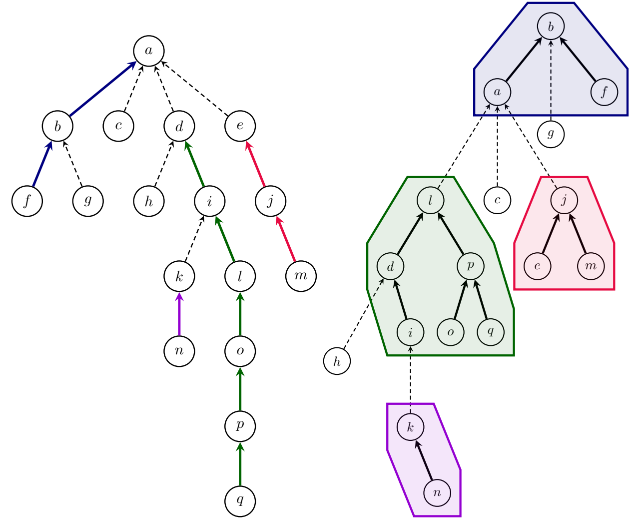

Figure 6. **Left**: Rooted tree modelled by LCTs, with its four preferred paths indicated by solid lines. The most-recently accessed path is $f \rightarrow b \rightarrow a$. **Right**: One possible configuration of LCT pointers which models the tree. Each preferred path is stored in a binary search tree (BST) *keyed by depth* (colour-coded to match the LHS figure), and path-parents (dashed) emanate from the root node of each BST—hence their source node may changed (e.g. $d \twoheadrightarrow a$ is represented as $l \twoheadrightarrow a$).

LCT operations Now we are ready to cover the specifics of how individual LCT operations (`find-root(u)`, `link(u, v)`, `cut(u)` and `evert(u)`) are implemented.

All of these operations rely on an efficient operation which *exposes* the path from a node u to its root, making it preferred—and making u the root of the entire LCT (i.e. the root node of the top-most BST). We will denote this operation as `expose(u)`, and assume its implementation is provided to us for now. As we will see, *all* of the interesting LCT operations will necessarily start with calls to `expose(u)` for nodes we are targeting.

Before discussing each of the LCT operations, note one important invariant after running `expose(u)`: node u is now the root node of the top-most BST (containing the nodes on the path from u to ρ_u), and *it has no right children* in this BST—as it is the deepest node in this path.

As in the main document, we will highlight in **blue** all changes to the ground-truth LCT pointers $\hat{\pi}_u$, which will be considered as the union of ground-truth BST parents $\hat{p}_u$ and path-parents $\hat{pp}_u$. Note that each node u will either have $\hat{p}_u$ or $\hat{pp}_u$; we will denote unused pointers with NIL. By convention, root nodes, ρ, of the entire LCT will point to themselves using a BST parent; i.e. $\hat{p}_\rho = \rho$, $\hat{pp}_\rho =$ NIL.

- `find-root(u)` can be implemented as follows: first

execute `expose(u)`. This guarantees that u is in the same BST as ρ_u, the root of the entire tree. Now, since the BST is keyed by depth and ρ_u is the shallowest node in the BST's preferred path, we just follow *left children* while possible, starting from u: ρ_u is the node at which this is no longer possible. We conclude with calling `expose` on ρ_u, to avoid pathological behaviour of repeatedly querying the root, accumulating excess complexity from following left children.

FIND-ROOT(u)
1 EXPOSE(u)
2 $\rho_u = u$
3 **while** $\text{left}_{\rho_u} \neq$ NIL
4 $\quad \rho_u = \text{left}_{\rho_u}$
5 EXPOSE(ρ_u)
6 **return** ρ_u

- `link(u, v)` has the precondition that u must be the root node of its respective tree (i.e. $u = \rho_u$), and u and v are not in the same tree. We start by running `expose(u)` and `expose(v)`. Attaching the edge $u \rightarrow v$ extends the preferred path from v to its root, ρ_v, to incorporate u. Given that u can have no left children in its BST (it is a root node, hence shallowest), this can be done simply by making v a left child of u (given v is shallower than u on the path $u \rightarrow v \rightarrow \cdots \rightarrow \rho_v$).

LINK(u, v)
1 EXPOSE(u)
2 EXPOSE(v)
3 $\text{left}_u = v$
4 $\hat{p}_v = u$

- `cut(u)`, as above, will initially execute `expose(u)`. As a result, u will retain all the nodes that are deeper than it (through path-parents pointed to by u), and can just be cut off from all shallower nodes along the preferred path (contained in u's left subtree, if it exists).

CUT(u)
1 EXPOSE(u)
2 **if** $\text{left}_u \neq$ NIL
3 $\quad \hat{p}_{\text{left}_u} = \text{left}_u$
4 $\quad \text{left}_u =$ NIL

- `evert(u)`, as visualised in Figure 5, needs to isolate the path from u to ρ_u, and *flip* the direction of all edges along it. The first part is already handled by calling `expose(u)`, while the second is implemented by recursively *flipping* left and right subtrees within the entire BST containing u (this makes shallower nodes in the path become deeper, and vice-versa).

This is implemented via *lazy propagation*: each node u stores a *flip bit*, ϕ_u (initially set to 0). Calling `evert(u)` will toggle node u's flip bit. Whenever we process node u, we further issue a call to a special operation, `release(u)`, which will perform any necessary flips of u's left and right children, followed by propagating the flip bit onwards. Note that `release(u)` *does not* affect parent-pointers $\hat{\pi}_u$—but it may affect outcomes of future operations on them.

RELEASE(u)
1 **if** $u \neq$ NIL **and** $\phi_u = 1$
2 $\quad$ SWAP($\text{left}_u, \text{right}_u$)
3 $\quad$ **if** $\text{left}_u \neq$ NIL
4 $\quad\quad \phi_{\text{left}_u} = \phi_{\text{left}_u} \oplus 1$
5 $\quad$ **if** $\text{right}_u \neq$ NIL
6 $\quad\quad \phi_{\text{right}_u} = \phi_{\text{right}_u} \oplus 1$
7 $\quad \phi_u = 0$

EVERT(u)
1 EXPOSE(u)
2 $\phi_u = \phi_u \oplus 1$
3 RELEASE(u)

Implementing `expose(u)` It only remains to provide an implementation for `expose(u)`, in order to specify the LCT operations fully.

LCTs use *splay trees* as the particular binary search tree implementation to represent each preferred path. These trees are also designed with "most-recent access" in mind: nodes recently accessed in a splay tree are likely to get accessed again, therefore any accessed node is turned into the *root node* of the splay tree, using the `splay(u)` operation. The manner in which `splay(u)` realises its effect is, in turn, via a sequence of complex *tree rotations*; such that `rotate(u)` will perform a rotation that brings u one level higher in the tree.

We describe these three operations in a *bottom-up* fashion: first, the lowest-level `rotate(u)`, which merely requires carefully updating all the pointer information. Depending on whether u is its parent's left or right child, a *zig* or *zag* rotation is performed—they are entirely symmetrical. Refer to Figure 7 for an example of a *zig* rotation followed by a *zag* rotation (often called *zig-zag* for short).

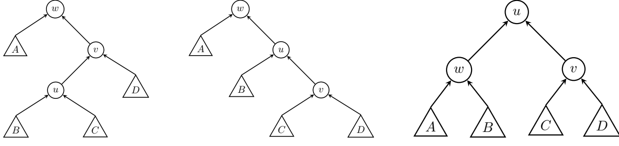

Figure 7. A schematic of a *zig-zag* rotation: first, node u is rotated around node v; then, node u is rotated around node w, bringing it two levels higher in the BST without breaking invariants.

ROTATE(u)

1 $v = \hat{p}_u$
2 $w = \hat{p}_v$
3 **if** $\text{left}_v = u$ // Zig rotation
4 $\quad\text{left}_v = \text{right}_u$
5 $\quad$**if** $\text{right}_u \neq$ NIL
6 $\quad\quad\hat{p}_{\text{right}_u} = v$
7 $\quad\text{right}_u = v$
8 **else** // Zag rotation
9 $\quad\text{right}_v = \text{left}_u$
10 $\quad$**if** $\text{left}_u \neq$ NIL
11 $\quad\quad\hat{p}_{\text{left}_u} = v$
12 $\quad\text{left}_u = v$
13 $\hat{pp}_u = \hat{pp}_v$
14 $\hat{p}_v = u$
15 $\hat{pp}_v =$ NIL
16 **if** $w \neq$ NIL // Adjust grandparent
17 $\quad$**if** $\text{left}_w = v$
18 $\quad\quad\text{left}_w = u$
19 $\quad$**else**
20 $\quad\quad\text{right}_w = u$
21 $\hat{p}_u = w$

Armed with the rotation primitive, we may define `splay(u)` as a repeated application of *zig*, *zig-zig* and *zig-zag* rotations, until node u becomes the root of its BST[4]. We also repeatedly perform *lazy propagation* by calling `release(u)` on any encountered nodes.

[4] Note: this exact sequence of operations is required to achieve optimal amortised complexity.

SPLAY(u)

1 **while** $\hat{p}_u \neq$ NIL // Repeat while u is not BST root
2 $\quad v = \hat{p}_u$
3 $\quad w = \hat{p}_v$
4 $\quad$RELEASE(w) // Lazy propagation
5 $\quad$RELEASE(v)
6 $\quad$RELEASE(u)
7 $\quad$**if** $w =$ NIL // *zig* rotation
8 $\quad\quad$ROTATE(u)
9 $\quad$**if** $(\text{left}_w = v) = (\text{left}_v = u)$ // *zig-zig* rotation
10 $\quad\quad$ROTATE(v)
11 $\quad\quad$ROTATE(u)
12 $\quad$**else** // *zig-zag* rotation
13 $\quad\quad$ROTATE(u)
14 $\quad\quad$ROTATE(u)
15 RELEASE(u) // In case u was root node already

Finally, we may define `expose(u)` as repeatedly interchanging calls to `splay(u)` (which will render u the root of its preferred-path BST) and appropriately following path-parents, $\hat{pp}_u$, to fuse u with the BST above. This concludes the description of the LCT operations.

EXPOSE(u)

1 **do**
2 $\quad$SPLAY(u) // Make u root of its BST
3 $\quad$**if** $\text{right}_u \neq$ NIL
4 $\quad\quad\hat{p}_{\text{right}_u} =$ NIL
5 $\quad\quad\hat{pp}_{\text{right}_u} = u$
6 $\quad\quad\text{right}_u =$ NIL
7 $\quad w = \hat{pp}_u$
8 $\quad$**if** $w \neq$ NIL // Attach u into w's BST
9 $\quad\quad$SPLAY(w)
10 $\quad\quad$**if** $\text{right}_w \neq$ NIL
11 $\quad\quad\quad\hat{p}_{\text{right}_w} =$ NIL
12 $\quad\quad\quad\hat{pp}_{\text{right}_w} = w$
13 $\quad\quad\text{right}_w = u$
14 $\quad\quad\hat{p}_u = w$
15 $\quad\quad\hat{pp}_u =$ NIL
16 **while** $\hat{p}_u \neq u$ // Repeat until u is root of its LCT

It is worth reflecting on the overall complexity of individual LCT operations, taking into account the fact they're propped up on `expose(u)`, which itself requires reasoning about *tree rotations*, followed by appropriately leveraging preferred path decompositions. This makes the LCT modelling task substantially more challenging than DSU.

Remarks on computational complexity and applications As can be seen throughout the analysis, the computational complexity of all LCT operations can be reduced to the computational complexity of calling `expose(u)`—adding only a constant overhead otherwise. `splay(u)` has a known amortised complexity of $O(\log n)$, for n nodes in

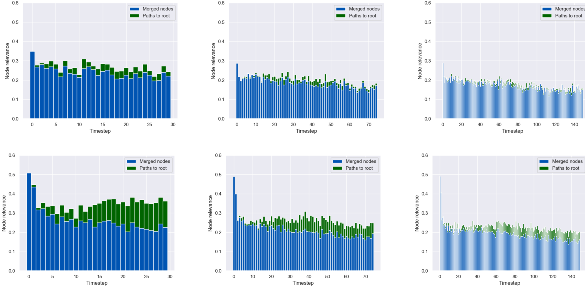

Figure 8. Credit assignment study results for the DSU setup, for the baseline GNN (**Top**) and the PGN (**Bottom**), arranged left-to-right by test graph size. PGNs learn to put larger emphasis on both the two nodes being operated on (**blue**) and the nodes on their respective paths-to-roots (**green**).

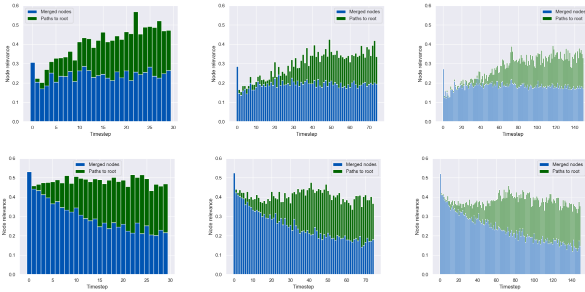

Figure 9. Credit assignment study results for the LCT setup, following same convention as Figure 8.

the BST; it seems that the ultimate complexity of exposing is this multiplied by the worst-case number of different preferred-paths encountered.

However, detailed complexity analysis can show that splay trees combined with preferred path decomposition yield an amortised time complexity of *exactly* $O(\log n)$ for *all* link/cut tree operations. The storage complexity is highly efficient, requiring $O(1)$ additional bookkeeping per node.

Finally, we remark on the utility of LCTs for performing *path aggregate* queries. When calling `expose(u)`, all nodes from u to the root ρ_u become exposed in the same BST, simplifying computations of important path aggregates (such as bottlenecks, lowest-common ancestors, etc). This can be augmented into an arbitrary `path(u, v)` operation by first calling `evert(u)` followed by `expose(v)`—this will expose only the nodes along the *unique* path from u to v within the same BST.

D. Credit assignment analysis

Firstly, recall how our decoder network, g, is applied to the latent state ($\vec{z}_i^{(t)}$, $\vec{h}_i^{(t)}$), in order to derive predicted query answers, $\vec{y}_i^{(t)}$ (Equation 3). Knowing that the *elementwise maximisation* aggregator performed the best as aggregation function, we can rewrite Equation 3 as follows:

$$\vec{y}^{(t)} = g\left(\max_i \vec{z}_i^{(t)}, \max_i \vec{h}_i^{(t)}\right) \tag{10}$$

This form of *max-pooling* readout has a unique feature: each dimension of the input vectors to g will be contributed to by *exactly* one node (the one which optimises the corresponding dimension in $\vec{z}_i^{(t)}$ or $\vec{h}_i^{(t)}$). This provides us with opportunity to perform a *credit assignment* study: we can verify how often every node has propagated its features into this vector—and hence, obtain a direct estimate of how "useful" this node is for the decision making by any of our considered models.

We know from the direct analysis of disjoint-set union (Section 3) and link/cut tree (Appendix C) operations that only a subset of the nodes are directly involved in decision-making for dynamic connectivity. These are exactly the nodes along the *paths* from u and v, the two nodes being operated on, to their respective *roots* in the data structure. Equivalently, these nodes directly correspond to the nodes tagged by ground-truth masks (nodes i for which $\hat{\mu}_i^{(t)} = 0$).

With the above hindsight, we compare a trained baseline GNN model against a PGN model, in terms of how much *credit* is assigned to these "important" nodes, throughout the rollout. The results of this study are visualised in Figures 8 (for DSU) and 9 (for LCT), visualising separately the credit assigned to the two nodes being operated on (**blue**) and the remaining nodes along their paths-to-roots (**green**).

From these plots, we can make several direct observations:

- In all settings, the PGN *amplifies* the overall credit assigned to these relevant nodes.
- On the DSU setup, the baseline GNN is likely suffering from *oversmoothing* effects: at larger test set sizes, it seems to hardly distinguish the paths-to-root (which are often very short due to path compression) from the remainder of the neighbourhoods. The PGN explicitly encodes the inductive bias of the structure, and hence more explicitly models such paths.
- As ground-truth LCT pointers are not amenable to path compression, paths-to-root may more significantly grow in lengthwith graph size increase. Hence at this point the oversmoothing effect is less pronounced for baselines; but in this case, LCT operations are highly centered on the node being operated on. The PGN learns to provide additional emphasis to the nodes operated on, u and v.

In all cases, it appears that through a careful and targeted constructed graph, the PGN is able to significantly overcome the oversmoothing issues with fully-connected GNNs,

providing further encouragement for applying PGNs in problem settings where strong credit assignment is required, one example of which are *search* problems.