

Algorithmic Execution via Graph Representation Learning

ScAi Lab Reading Group Report



Zhiping (Patricia) Xiao
University of California, Los Angeles

October 13, 2021

Introduction

Neural Execution of Graph Algorithms

Pointer Graph Networks

More Related Works

Introduction



Petar's work:

- ▶ Neural Execution of Graph Algorithms (ICLR'20)
- ▶ Pointer Graph Networks (NeurIPS'20)

Author's Presentations:

- ▶ [https://slideslive.com/38938392/
algorithmic-reasoning-in-the-real-world](https://slideslive.com/38938392/algorithmic-reasoning-in-the-real-world)¹
- ▶ <https://petar-v.com/talks/Algo-WWW.pdf>
- ▶ (and more:
<https://petar-v.com/communications.html>)

¹Special thanks to Ziniu.

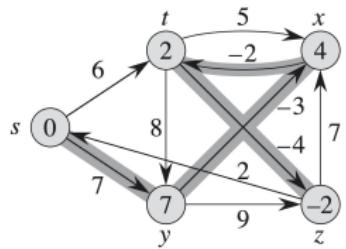


Figure: Algorithms

- Inputs must match spec
- Not robust to task variations
- + Interpretable operations
- + Trivially strongly generalise
- + Small data is fine

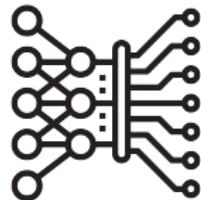


Figure: Neural Networks

- + Operate on raw inputs
- + Models are reusable across tasks
- Lack of interpretability
- Unreliable when extrapolating
- Require big data

Scenario 1: Parallel Algorithm

Many algorithms share subroutines. e.g.:

- ▶ Shortest-Path Computation via Bellman-Ford Algorithm
- ▶ Reachability Computation via Breadth-First Search

both enumerates sets of edges adjacent to a particular node.

Scenario 2: Sequential Algorithm

Some Algorithms focus on one node at a time (different than ↑).

e.g.:

- ▶ Minimum Spanning Trees generation via Prim's Algorithm

So far, researchers have studied: use *ground-truth* algorithmic solution (algorithm) to drive learning (neural networks).

Petar's works: use neural networks (**graph** neural networks) to execute classical algorithms (on **graphs**).

They name it as **Neural Graph Algorithm Execution**.

The approach that:

- ▶ Learn several algorithms simultaneously
- ▶ Provide a supervision signal
 - ▶ signal: driven by prior knowledge on how classical algorithms' behaviors

and thus transfer knowledge between different algorithms.

Neural Execution of Graph Algorithms



Two roles:

- ▶ Part of the problem provided;
- ▶ Inputs to a GNN.

The graph $G = (V, E)$ consists of:

- ▶ V : the set of nodes / vertices;
- ▶ E : the set of edges / node-pairs.

GNN receives a sequence of T graph-structured inputs (index $t \in \{1, \dots, T\}$),

- ▶ Each node $i \in V$ has features $\mathbf{x}_i^{(t)} \in \mathbb{R}^{N_x}$
- ▶ Each edge $(i, j) \in E$ has features $\mathbf{e}_{ij}^{(t)} \in \mathbb{R}^{N_e}$
- ▶ Each step *node-level* output $\mathbf{y}_i^{(t)} \in \mathbb{R}^{N_y}$

Consisting of three components:

- ▶ an *encoder* network f_A for **each** algorithm A
 - ▶ inputs: node feature \mathbf{x} , (previous) latent feature \mathbf{h}
 - ▶ output: encoded input \mathbf{z}
- ▶ a *processor* network P shared among all algorithms
 - ▶ inputs: edge feature \mathbf{e} , encoded input \mathbf{z}
 - ▶ output: latent feature \mathbf{h}
- ▶ a *decoder* network g_A for **each** algorithm A
 - ▶ inputs: encoded input \mathbf{z} , latent feature \mathbf{h}
 - ▶ output: node-level outputs \mathbf{y}

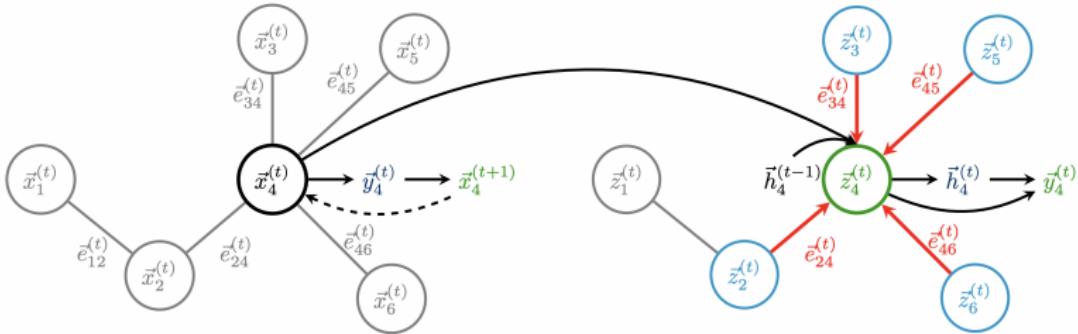


Figure: Relation between local computation of graph algorithm (**left**) and the neural graph algorithm executor (**right**).

Node values $\mathbf{y}_i^{(t)}$ (e.g. reachability, shortest-path distance, etc.) are updated at every step of execution.

Analogously, node values are predicted by the neural executor from hidden rep $\mathbf{h}_i^{(t)}$ via message-passing.

(Figure 1 of the paper.)

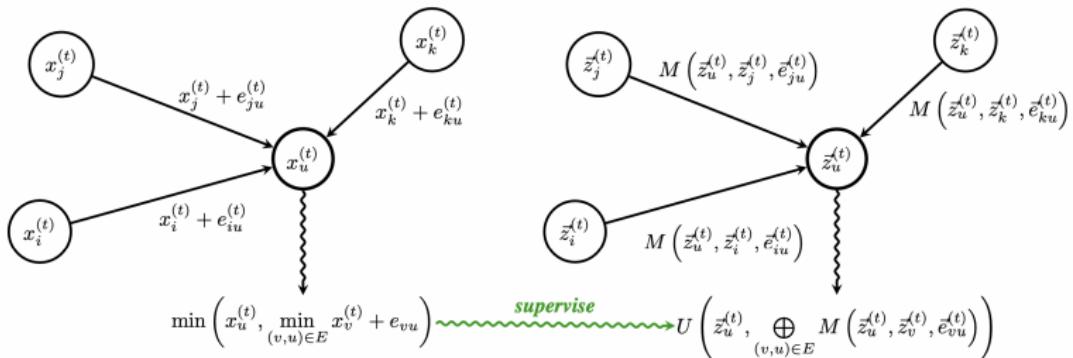


Figure: An example. Illustrating the alignment of one step of the Bellman-Ford algorithm (**left**) with one step of a message passing neural network (**right**), and the supervision signal used for the algorithm learner.
 (Figure 2 of the paper.)

From features to encoded inputs:

- ▶ $\mathbf{x}_i^{(t)}$: node feature of node i at step t
- ▶ $\mathbf{h}_i^{(t-1)}$: *previous* latent feature of node i
- ▶ $\mathbf{z}_i^{(t)}$: encoded input of node i at step t

$$\mathbf{z}_i^{(t)} = f_A(\mathbf{x}_i^{(t)}, \mathbf{h}_i^{(t-1)}) , \quad \mathbf{h}_i^{(0)} = 0$$

From encoded inputs to latent representation:

- ▶ $\mathbf{E}^{(t)} = \{\mathbf{e}_{ij}^{(t)}\}_{(i,j) \in E}$: all edge features at step t
- ▶ $\mathbf{Z}^{(t)} = \{\mathbf{z}_i^{(t)}\}_{i \in V}$: all encoded inputs at step t
- ▶ $\mathbf{H}^{(t)} = \{h_i^t \in \mathbb{R}^K\}_{i \in V}$: all latent features at step t

$$\mathbf{H}^{(t)} = P(\mathbf{Z}^{(t)}, \mathbf{E}^{(t)})$$

Note that:

1. Parameters of P are **shared** among all algorithms being learnt.
2. P make decision on when to terminate the algorithm, handled by an *algorithm-specific* termination network T_A

T_A is specific to algorithm A :

- ▶ $\mathbf{H}^{(t)} = \{h_i^t \in \mathbb{R}^K\}_{i \in V}$: all latent features at step t
- ▶ $\overline{\mathbf{H}^{(t)}} = \frac{1}{|V|} \sum_{i \in V} \mathbf{h}_i^{(t)}$: the average node embedding at step t
- ▶ σ : the logistic sigmoid activation
- ▶ $\tau^{(t)}$: the probability of termination

$$\tau^{(t)} = \sigma(T_A(\mathbf{H}^{(t)}, \overline{\mathbf{H}^{(t)}}))$$

Only when $\tau^{(t)}$ is below some threshold (e.g. 0.5) we will move on to the next step ($t + 1$).

From (algorithm-specific) encoded inputs, and shared latent features, to algorithm-specific outputs:

- ▶ $\mathbf{z}_i^{(t)}$: encoded input of node i at step t
- ▶ $\mathbf{h}_i^{(t)}$: latent feature of node i at step t
- ▶ $\mathbf{y}_i^{(t)}$: algorithm-specific output of node i at step t

$$\mathbf{y}_i^{(t)} = g_A(\mathbf{z}_i^{(t)}, \mathbf{h}_i^{(t)})$$

If the algorithm hasn't been terminated ($\tau^{(t)}$ is big enough), parts of $\mathbf{y}_i^{(t)}$ might be reused in $\mathbf{x}_i^{(t+1)}$ (next step node feature).

All algorithms need to be executed simultaneously.

- ▶ Make processor network P algorithm-agnostic.

The majority of the representational power should be placed in the processor network P .

- ▶ All the algorithm-dependent networks f_A , g_A , T_A are simply linear projections.

Most algorithms require making *discrete decisions* over neighborhoods (e.g. “which edge to take”).

- ▶ *Message-passing neural network* with a *maximization aggregator* is naturally suitable.

GATs (Graph Attention Networks):

$$\mathbf{h}_i^{(t)} = \text{ReLU} \left(\sum_{(j,i) \in E} \alpha(\mathbf{z}_i^{(t)}, \mathbf{z}_j^{(t)}, \mathbf{e}_{ij}^{(t)}) \mathbf{W} \mathbf{z}_j^{(t)} \right),$$

where W is learnable projection matrix, α is the attention mechanism producing *scalar coefficients*.

MPNNs (Message-Passing Neural Networks):

$$\mathbf{h}_i^{(t)} = U \left(\mathbf{z}_i^{(t)}, \bigoplus_{(j,i) \in E} M(\mathbf{z}_i^{(t)}, \mathbf{z}_j^{(t)}, \mathbf{e}_{ij}^{(t)}) \right),$$

where M, U are neural networks producing *vector messages*. \bigoplus represents an element-wise aggregation operator, could be maximization, summation, averaging, etc.

Employ a GNN layer as P , using MPNNs:

$$\mathbf{h}_i^{(t)} = U \left(\mathbf{z}_i^{(t)}, \bigoplus_{(j,i) \in E} M \left(\mathbf{z}_i^{(t)}, \mathbf{z}_j^{(t)}, \mathbf{e}_{ij}^{(t)} \right) \right),$$

- ▶ Inserting a self-edge to every node, to make retention of self-information easier.
- ▶ M, U : linear projections
- ▶ \bigoplus : try mean, sum, max
- ▶ Compare to GATs baselines

Graphs are generated.³

For each edge, $\mathbf{e}_{ij}^{(t)} \in \mathbb{R}$ is simply a real-value weight, drawn uniformly from range [0.2, 1].

- ▶ Benefit: randomly-sampled edge weights guarantees the uniqueness of the recovery solution, simplifying downstream evaluation.

³Follows You et al. 2018, 2019.

Both algorithms:

1. Initialize by randomly select a source node s
2. Input $x_i^{(1)}$ is initialized according to $i = s$ or $i \neq s$
3. Aggregate neighborhood information to update
4. Requires discrete decisions (which edge to select)
 - ▶ For the baselines e.g. GAT, coefficients are thus *sharpened*.

BFS (Breadth-First Search) for reachability:

$$x_i^{(1)} = \begin{cases} 1 & i = s \\ 0 & i \neq s \end{cases}$$

$$x_i^{(t+1)} = \begin{cases} 1 & x_i^{(t)} = 1 \\ 1 & \exists j. (j, i) \in E \wedge x_j^{(t)} = 1 \\ 0 & \text{otherwise} \end{cases}$$

$x_i^{(t)}$: is i reachable from s in $\leq t$ hops?

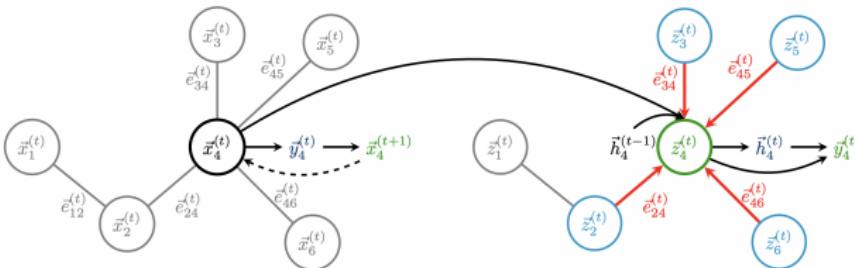
Bellman-Ford for Shortest Paths:

$$x_i^{(1)} = \begin{cases} 0 & i = s \\ +\infty & i \neq s \end{cases}$$

$$x_i^{(t+1)} = \min \left(x_i^{(t)}, \min_{(j,i) \in E} x_j^{(t)} + e_{ji}^{(t)} \right)$$

$x_i^{(t)}$: shortest distance from s to i (using $\leq t$ hops)

Recall:



For BFS, no additional information is being computed, thus node-level output $y_i^{(t)} = x_i^{(t+1)}$

For Bellman-Ford, one have to remember the *predecessor* so as to reconstruct the path. Therefore, $y_i^{(t)} = p_i^{(t)} || x_i^{(t+1)}$ where

$$\text{predecessor } p_i^t = \begin{cases} i & i = s \\ \arg \min_{j; (j,i) \in E} x_j^{(t)} + e_{ji}^{(t)} & i \neq s \end{cases}$$

Prim's Algorithm for Minimum Spanning Trees (MST):

$$x_i^{(1)} = \begin{cases} 1 & i = s \\ 0 & i \neq s \end{cases}$$

$$x_i^{(t+1)} = \begin{cases} 1 & x_i^{(t)} = 1 \\ 1 & i = \arg \min_j \text{ s.t. } x_j^{(t)} = 0 \min_k \text{ s.t. } x_k^{(t)} = 1 e_{jk}^{(t)} \\ 0 & \text{otherwise} \end{cases}$$

$x_i^{(t)}$: is i in the partial MST tree built from s after t steps?

Similar to Bellman-Ford, the *predecessor* has to be recorded.
Keeping $p_i^{(t)}$ — the predecessor of i in the partial MST.

Experimental Results

Trained on a graph of 20 nodes, performing well on graphs with more nodes.

Table 1: Accuracy of predicting reachability at different test-set sizes, trained on graphs of 20 nodes. GAT* correspond to the best GAT setup as per Section 3 (GAT-full using the full graph).

Model	Reachability (mean step accuracy / last-step accuracy)		
	20 nodes	50 nodes	100 nodes
LSTM (Hochreiter & Schmidhuber, 1997)	81.97% / 82.29%	88.35% / 91.49%	68.19% / 63.37%
GAT* (Veličković et al., 2018)	93.28% / 99.86%	93.97% / 100.0%	92.34% / 99.97%
GAT-full* (Vaswani et al., 2017)	78.40% / 77.86%	85.76% / 91.83%	88.98% / 91.51%
MPNN-mean (Gilmer et al., 2017)	100.0% / 100.0%	61.05% / 57.89%	27.17% / 21.40%
MPNN-sum (Gilmer et al., 2017)	99.66% / 100.0%	94.25% / 100.0%	94.72% / 98.63%
MPNN-max (Gilmer et al., 2017)	100.0% / 100.0%	100.0% / 100.0%	99.92% / 99.80%

Table 3: Mean squared error for predicting the intermediate distance information from Bellman-Ford, and accuracy of the termination network compared to the ground-truth algorithm, averaged across all timesteps. (*curriculum*) corresponds to a curriculum wherein reachability is learnt first. (*no-reach*) corresponds to training without the reachability task.

Model	B-F mean squared error / mean termination accuracy		
	20 nodes	50 nodes	100 nodes
LSTM (Hochreiter & Schmidhuber, 1997)	3.857 / 83.43%	11.92 / 86.74%	74.36 / 83.55%
GAT* (Veličković et al., 2018)	43.49 / 85.33%	123.1 / 84.88%	183.6 / 82.16%
GAT-full* (Vaswani et al., 2017)	7.189 / 77.14%	28.89 / 75.51%	58.08 / 77.30%
MPNN-mean (Gilmer et al., 2017)	0.021 / 98.57%	23.73 / 89.29%	91.58 / 86.81%
MPNN-sum (Gilmer et al., 2017)	0.156 / 98.09%	4.745 / 88.11%	+∞ / 87.71%
MPNN-max (Gilmer et al., 2017)	0.005 / 98.89%	0.013 / 98.58%	0.238 / 97.82%
MPNN-max (<i>curriculum</i>)	0.021 / 98.99%	0.351 / 96.34%	3.650 / 92.34%
MPNN-max (<i>no-reach</i>)	0.452 / 80.18%	2.512 / 91.77%	2.628 / 85.22%

Table 2: Accuracy of predicting the shortest-path predecessor node at different test-set sizes. (*curriculum*) corresponds to a curriculum wherein reachability is learnt first. (*no-reach*) corresponds to training without the reachability task. (*no-algo*) corresponds to the classical setup of directly training on the predecessor, without predicting any intermediate outputs or distances.

Model	Predecessor (mean step accuracy / last-step accuracy)		
	20 nodes	50 nodes	100 nodes
LSTM (Hochreiter & Schmidhuber, 1997)	47.20% / 47.04%	36.34% / 35.24%	27.59% / 27.31%
GAT* (Veličković et al., 2018)	64.77% / 60.37%	52.20% / 49.71%	47.23% / 44.90%
GAT-full* (Vaswani et al., 2017)	67.31% / 63.99%	50.54% / 48.51%	43.12% / 41.80%
MPNN-mean (Gilmer et al., 2017)	93.83% / 93.20%	58.60% / 58.02%	44.24% / 43.93%
MPNN-sum (Gilmer et al., 2017)	82.46% / 80.49%	54.78% / 52.06%	37.97% / 37.32%
MPNN-max (Gilmer et al., 2017)	97.13% / 96.84%	94.71% / 93.88%	90.91% / 88.79%
MPNN-max (<i>curriculum</i>)	95.88% / 95.54%	91.00% / 88.74%	84.18% / 83.16%
MPNN-max (<i>no-reach</i>)	82.40% / 78.29%	78.79% / 77.53%	81.04% / 81.06%
MPNN-max (<i>no-algo</i>)	78.97% / 95.56%	83.82% / 85.87%	79.77% / 78.84%

Table 6: Accuracy of selecting the next node to add to the minimum spanning tree, and predicting the minimum spanning tree predecessor node—at different test-set sizes. (*no-algo*) corresponds to the classical setup of directly training on the predecessor, without adding nodes sequentially.

Model	Accuracy (next MST node / MST predecessor)		
	20 nodes	50 nodes	100 nodes
LSTM (Hochreiter & Schmidhuber, 1997)	11.29% / 52.81%	3.54% / 47.74%	2.66% / 40.89%
GAT* (Veličković et al., 2018)	27.94% / 61.74%	22.11% / 58.66%	10.97% / 53.80%
GAT-full* (Vaswani et al., 2017)	29.94% / 64.27%	18.91% / 53.34%	14.83% / 51.49%
MPNN-mean (Gilmer et al., 2017)	90.56% / 93.63%	52.23% / 88.97%	20.63% / 80.50%
MPNN-sum (Gilmer et al., 2017)	48.05% / 77.41%	24.40% / 61.83%	31.69% / 43.98%
MPNN-max (Gilmer et al., 2017)	87.85% / 93.23%	63.89% / 91.14%	41.37% / 90.02%
MPNN-max (<i>no-algo</i>)	— / 71.02%	— / 49.83%	— / 23.61%

The tasks in this paper only focus on node-level representation (due to the requirement of the experiments).

In theory, this model could also easily include:

- ▶ edge-level outputs;
- ▶ graph-level inputs / outputs.

Not considering corner-case inputs (e.g. negative weight cycles).

Pointer Graph Networks



The previous work make GNNs learn graph algorithms, and transfer between them (MTL), using a single neural core (Process Network P) capable of: sorting, path-finding, binary addition.

PGNs is a framework that further **expands** the space of general-purpose algorithms that can be neurally executed.

Similar yet different. Different data structure:

- ▶ Previous: sequence of graphs $G = (V, E)$
- ▶ PGNs: sequence of pointer-based structures, pointer adjacency matrix $\Pi^{(t)} \in \mathbb{R}^{n \times n}$ is dynamic (like (V, Π))

Problem setup is different. PGN:

- ▶ A sequence of operation inputs (of n entities at each step):

$$\mathcal{E}^{(t)} = \{\mathbf{e}_1^{(t)}, \mathbf{e}_2^{(t)}, \dots, \mathbf{e}_n^{(t)}\},$$

$\mathbf{e}_i^{(t)}$ represents feature of entity i at time t , denoting some operation (add / remove edge etc.).

- ▶ Problem: predicting target outputs $\mathbf{y}_i^{(t)}$ from $\mathcal{E}^{(1)}, \dots, \mathcal{E}^{(t)}$

Tasks on *Dynamic Graph Connectivity* are used to illustrate the benefits of PGNs in the paper.

- ▶ DSU: disjoint-set unions, incremental graph connectivity
- ▶ LCT: link/cut trees, fully dynamic tree connectivity

Following the encoder-process-decoder paradigm on a sequence ($t = 1, \dots, T$) graph-structured inputs $G = (V, E)$:

- ▶ an *encoder* network f_A for **each** A : $\mathbf{X}^{(t)}, \mathbf{H}^{(t-1)} \rightarrow \mathbf{Z}^{(t)}$
 - ▶ $\mathbf{z}_i^{(t)} = f_A(\mathbf{x}_i^{(t)}, \mathbf{h}_i^{(t-1)})$, $\mathbf{h}_i^{(0)} = 0, i \in V$
 - ▶ implemented as linear projections
- ▶ a *processor* network P (**shared**): $\mathbf{Z}^{(t)}, \mathbf{E}^{(t)} \rightarrow \mathbf{H}^{(t)}$
 - ▶ $\mathbf{H}^{(t)} = P(\mathbf{Z}^{(t)}, \mathbf{E}^{(t)})$
 - ▶ implemented as MPNNs
- ▶ a *decoder* network g_A for **each** A : $\mathbf{Z}^{(t)}, \mathbf{H}^{(t)} \rightarrow \mathbf{Y}^{(t)}$
 - ▶ $\mathbf{y}_i^{(t)} = g_A(\mathbf{z}_i^{(t)}, \mathbf{h}_i^{(t)})$
 - ▶ implemented as linear projections

Also encoder-process-decoder paradigm, on sequence of pointer-based inputs: $\mathcal{E}^{(t)} = \{\mathbf{e}_i^{(t)}\}_{i=1}^n$, pointer adjacency matrix $\boldsymbol{\Pi}^{(t)} \in \mathbb{R}^{n \times n}$:

- ▶ an *encoder* network $f: \mathcal{E}^{(t)}, \mathbf{H}^{(t-1)} \rightarrow \mathbf{Z}^{(t)}$
 - ▶ $\mathbf{z}_i^{(t)} = f(\mathbf{e}_i^{(t)}, \mathbf{h}_i^{(t-1)})$, $\mathbf{h}_i^{(0)} = 0, i \in \{1, \dots, n\}$
 - ▶ implemented as linear projections
- ▶ a *processor* network $P: \mathbf{Z}^{(t)}, \boldsymbol{\Pi}^{(t-1)} \rightarrow \mathbf{H}^{(t)}$
 - ▶ $\mathbf{H}^{(t)} = P(\mathbf{Z}^{(t)}, \boldsymbol{\Pi}^{(t-1)})$
 - ▶ implemented as MPNNs
- ▶ a *decoder* network $g: \mathbf{Z}^{(t)}, \mathbf{H}^{(t)} \rightarrow \mathbf{Y}^{(t)}$
 - ▶ $\mathbf{y}^{(t)} = g(\bigoplus_i \mathbf{z}_i^{(t)}, \bigoplus_i \mathbf{h}_i^{(t)})$
 - ▶ \bigoplus : permutation-invariant aggregator (e.g. sum / max)
 - ▶ implemented as linear projections

Inductive Bias: Many efficient algorithms only modify a small subset of the entities at once.

To incorporate it: Introducing masking $\mu_i^{(t)} \in \{0, 1\}$ for each node at each step,

$$\mu_i^{(t)} = \mathbb{I}_{\psi(\mathbf{z}_i^{(t)}, \mathbf{h}_i^{(t)}) > 0.5},$$

where ψ is the *masking network*, implemented as *linear* layers of appropriate dimensionality, with output activation being logistic sigmoid (enforcing probabilistic interpretation).

$$\Pi_{ij}^{(t)} = \tilde{\Pi}_{ij}^{(t)} \vee \tilde{\Pi}_{ji}^{(t)},$$

where it is found that symmetrise the matrix is beneficial, and $\tilde{\Pi}^{(t)}$ denotes the pointers before symmetrisation.

$$\tilde{\Pi}_{ij}^{(t)} = \mu_i^{(t)} \tilde{\Pi}_{ij}^{(t-1)} + (1 - \mu_i^{(t)}) \mathbb{I}_{j=\arg\max_k(\alpha_{ik}^{(t)})},$$

where μ_i are the sparsity mask we've mentioned before, $(1 - \mu_i^{(t)})$ is negating the mask. α is self-attention coefficient of $\mathbf{h}_i^{(t)}$:

$$\alpha_{ik}^{(t)} = \text{softmax}_j \left(\langle \mathbf{W}_q \mathbf{h}_i^{(t)}, \mathbf{W}_k \mathbf{h}_i^{(t)} \rangle \right)$$

where \mathbf{W}_q and \mathbf{W}_k are learnable linear transformations.
i.e. Nodes i, j are linked together ($\Pi_{ij}^{(t)} = 1$) if they are (1)
selected by the sparse mask (2) the most relevant to each other.

In the previous work, P using MPNNs with U, M being linear layers with ReLU activation functions:

$$\mathbf{h}_i^{(t)} = U \left(\mathbf{z}_i^{(t)}, \bigoplus_{(j,i) \in E} M(\mathbf{z}_i^{(t)}, \mathbf{z}_j^{(t)}, \mathbf{e}_{ij}^{(t)}) \right).$$

In PGNs, P is also using MPNN with linear U, M with ReLU.

$$\mathbf{h}_i^{(t)} = U \left(\mathbf{z}_i^{(t)}, \bigoplus_{\Pi_{ji}^{(t-1)}=1} M(\mathbf{z}_i^{(t)}, \mathbf{z}_j^{(t)}) \right),$$

where among all possible choices of aggregator \bigoplus , once again, (element-wise) max outperforms the rest.

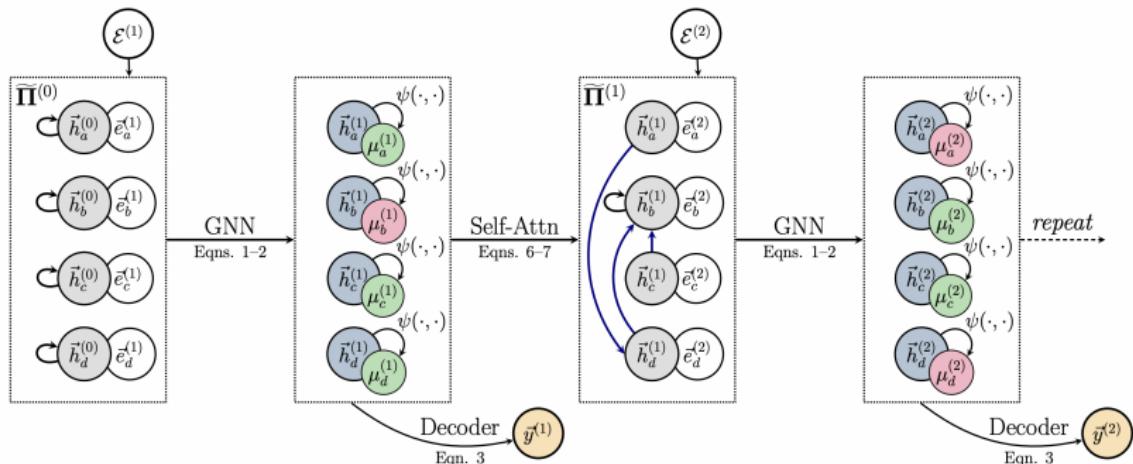


Figure: Visualization of pointer graph network (PGN) dataflow.
(Figure 1 in the paper.)

PGNs consider loss of three components at the same time:

- ▶ The downstream query loss in $\mathbf{y}^{(t)}$ prediction
- ▶ Difference between $\alpha^{(t)}$ and ground-truth pointers $\hat{\mathbf{\Pi}}^{(t)}$ (cross-entropy)
- ▶ Output from masking network ψ compared to ground-truth modification at time step t (binary cross-entropy)

Thereby, domain knowledge is introduced while training.

DSU: disjoint-set unions

QUERY-UNION(u, v) is called each step t , specified by

$$\mathbf{e}_i^{(t)} = r_i || \mathbb{I}_{i=u \vee i=v},$$

- ▶ r_i : priority of node i
- ▶ $\mathbb{I}_{i=u \vee i=v}$: is node i being operated on?
- ▶ $\hat{\mathbf{y}}^{(t)}$: u, v in the same set?
- ▶ $\hat{\mu}_i^{(t)}$: node i visible by FIND(u) or FIND(v)?
- ▶ $\hat{\Pi}_{ij}^{(t)}$: $\hat{\pi}_i = j$ after executing?

LCT: link/cut trees

QUERY-TOGGLE(u, v) is called each step t , specified by

$$\mathbf{e}_i^{(t)} = r_i || \mathbb{I}_{i=u \vee i=v},$$

- ▶ r_i : priority of node i
- ▶ $\mathbb{I}_{i=u \vee i=v}$: is node i being operated on?
- ▶ $\hat{\mathbf{y}}^{(t)}$: u, v connected?
- ▶ $\hat{\mu}_i^{(t)}$: node i visible while executing?
- ▶ $\hat{\Pi}_{ij}^{(t)}$: $\hat{\pi}_i = j$ after executing?

More Related Works



More keywords: program synthesis, learning to execute, message-passing neural network, neural execution engines, etc.

Important previous works:

- ▶ Neural Programmer-Interpreters (ICLR'16)
- ▶ Deep Sets (NeurIPS'17)

Application to reinforcement learning:

- ▶ XLVIN: eXecuted Latent Value Iteration Nets (NeurIPS'20 Workshop)

Thank You! 😊