Improving Out-Of-Distribution Generalization of Neural Algorithmic Reasoning Tasks

Sadegh Mahdavi
22542575

Abstract

Artificial Neural Networks have been successful in solving many tasks ranging from natural language processing to computer vision. Yet, their performance is still disappointing when training them to execute algorithmic tasks such as sorting, breadth first search (BFS), and depth first search (DFS). This low performance exacerbated when these networks are given test inputs of sizes larger than that of the training data. In this project, we leverage Graph Neural Networks and propose a method to improve the out-of-distribution generalization of neural algorithmic reasoning tasks. Our key improvements are using a bidirectional graph neural network and appending random features to the underlying graph. We evaluate our method on the CLRS benchmark containing 30 algorithmic tasks and report state-of-the art performance on this benchmark with 10.37%average improvement over the previous methods.

1 Introduction

Algorithms have been a vital part of today's computing systems, solving tasks ranging from scheduling to finding shortest path in a real-world map. Algorithms come with several benefits, such as performance guarantees, applicability to several tasks without any significant change, and transferability between tasks. Using them to solve tasks, however, is not straightforward in several situations. For instance, for any task, a computer scientist has to come up with some algorithm to perform a task and the process is hand-engineered. Moreover, to apply an algorithm on data, the raw data needs to be processed and turned into an abstract form, which might lead to errors (e.g., turning a raw image into a graph with edges estimated from a noisy data). Finally, some algorithms are lie in the NP-Hard class and require exponential amount of time for execution, which makes it impossible to execute on medium to large inputs. Coming up with run-time efficient approximate algorithms is also challenging and sometimes requires several years of research. On the other hand, deep learning has not only been successful in solving tasks of several domains, but also does not suffer from the mentioned issues. A neural network contains almost zero hand-engineered features. For instance, a transformer [3] could solve tasks of several domains with minor changes to its architecture. Moreover, such networks are famous for their abilities in performing tasks end-to-end on raw noisy data. These networks are also great function-approximators, having the ability to approximate any function to any precision, given large enough capacity.

As a result, leveraging neural networks to mimic algorithms would come with several benefits of both worlds: performance guarantees of algorithms, end-to-end training of neural networks, function approximation of neural networks for time-complexity infeasible algorithms, and benefiting from GPUs and TPUs for faster parallel execution of algorithms. Recently, researchers have been working on this task and calling it Neural Algorithmic Reasoning [4].

Previous works have shown that performing such tasks is challenging, especially when the model is evaluated on a test dataset with larger size than train dataset (i.e., fails to generalize to out-of-distribution data) [6]. For example, while deep neural networks could be trained to perform sorting on sequences of a given length, when faced with sequences of four times larger length, they fail badly and poorly generalize. In this project, we seek to improve this out-of-distribution (OOD) generalization of such tasks. To this end, we will propose a model with a graph neural network architecture that acts as a neural algorithmic executor to execute algorithmic reasoning tasks. Finally, we show that our model outperforms the baselines on the CLRS[6] benchmark which contains several algorithmic reasoning tasks inspired by algorithms of the famous CLRS book.

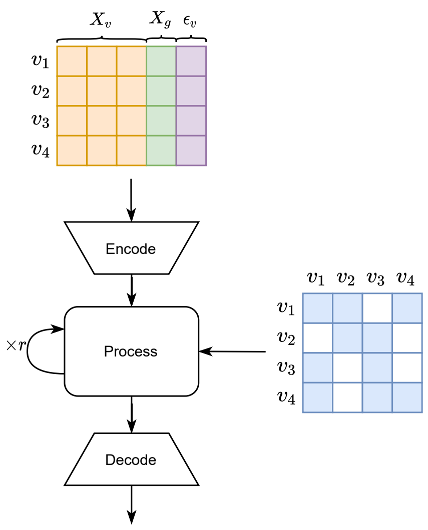
2 Related Work

Recently, several works have been proposed to tackle challenges on the intersection of algorithms and machine learning. [10] studied the relation between neural network architecture decision and the accuracy on the underlying algorithmic task. [11] proposed a tweaked variant of transformers to perform subroutines using neural networks. Although this work puts a step forward towards using neural networks as computational subroutines and perhaps enabling execution of algorithms on GPUs and TPUs (instead of CPUs), there is still the need for hand-engineering and putting together several pieces of these algorithmic engines. Moving towards a more general-purpose method, [5] proposed a framework to mimic individual steps of algorithms using an algorithm-agnostic processor. Despite proposing a general recipe for all kinds of algorithms, the processor introduced in the paper operates on intermediate steps of the algorithms, therefore making it challenging to apply it end-to-end, and when it is applied end-to-end, the results on larger size inputs are not promising. The authors of [7] proposed an architecture that operates on a selective set of nodes and hence brings sparsity, but again, their approach is based on supervision on intermediate steps of the algorithm.

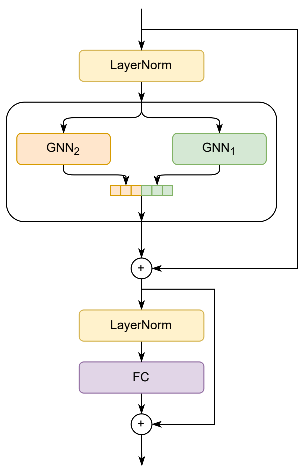
In search of a comprehensive and unified benchmark for comparing proposed methods on algorithmic reasoning tasks, the authors of [6] proposed a benchmark containing 30 algorithmic tasks, ranging from graph-based tasks such as BFS and Dijkstra, to scheduling and sorting tasks of sequences. According to the baseline results reported in the benchmark, current well-known graph neural network-based tasks lack the inductive bias of performing such algorithms and show poor results on OOD data. This is the most relevant work to ours, since our setting is similar to theirs and we seek to find a more expressive processor that could tackle the challenge of OOD generalization on this benchmark. The baseline of the benchmark utilizes intermediate steps of algorithms (calling them *hints*) in the training phase, and number of underlying intermediate steps at test time. This setting is restrictive since (1) not all algorithms contain reasonable hints or number of hints (e.g., NP-Hard algorithms) (2) when translating sequential algorithms to a parallel message-passing framework, the hints become implementation-dependent and are no longer uniquely determined. This might make the task of mimicing an implementation-specific version of an algorithm even harder than the actual algorithm itself, especially for inefficient sequential algorithms which leads to oversmoothing issues in GNNs[12] (3) knowing number of hints at test-time is not a realistic assumption (4) enforcing inefficient hints prevents the processing unit from benefiting from the full power of message-passing (e.g., forcing the processor to perform an algorithm step-by-step in $2n$ steps while it could have been done intelligently in n steps). Hence, we focus on proposing an end-to-end trainable neural network without utilizing any hints, as well as using an algorithm-independent number of processing steps. Although, our setting comes at the cost of not being able to distinguish between algorithms of the same output (e.g., between merge-sort, quick-sort, and bubble-sort).

3 Method

We address all algorithmic problems through the lens of graphs. That is, for sequences we assume a fully-connected underlying graph and for graph structured problems, we consider the graph adjacency matrix itself. This allows us to benefit from the rich literature on GNNs and solve problems in a universal manner. So, for any reasoning task, we assume a (possibly fully connected) graph G with n nodes $V = \{v_1, v_2, \ldots, v_n\}$ and m edges $E = \{e_i = (v_{i_1}, v_{i_2}) : v_{i_1}, v_{i_2} \in V, 1 \leq i \leq m\}$, adjacency matrix $A \in \mathbb{R}^{n \times n}$, and edge features. Moreover, define $\mathcal{N}(v_i)$ as the set of neighboring nodes of v_i. Also, for each node v_i we assume a feature vector $x_{v_i} \in \mathbb{R}^{d_v}$ (e.g., node index), for each edge e_i a feature vector $x_{e_i} \in \mathbb{R}^{d_e}$ (e.g., edge weight), and finally a graph-level feature vector $x_g \in \mathbb{R}^{d_g}$ for graph g (e.g., target scalar for binary search tree). We indicate output of the task by y which could be a mask/pointer/categorical node/edge/graph level property depending on the task (e.g., pointer node level for the sorting tasks where each node points to its successor). Taking Breadth-first search (BFS) algorithm as an example, the input will be an undirected graph, as well as a node index



(a) Overall architecture



(b) Processor architecture

Figure 1: The architecture of our method. (a) shows the overall encode-decode-process architecture; node features $\mathcal{X}_v$, graph features $\mathcal{X}_g$, and random features ϵ_v are concatenated and encoded as node features. Edge features are fed to the processor at each recurrent step. The outputs are created after r recursive steps using the decoder block (b) shows the processor architecture; similar to a pre-norm transformer, but GNN layer instead of attention layer.

for the starting node. The output would be a pointer for each node, pointing to its parent in the BFS algorithm. Therefore, set of nodes V will be the set of nodes of the underlying graph, matrix A the symmetric adjacency matrix of the undirected graph, x_{v_i} a pair $(Index_{v_i}, IsStart_{v_i})$ for each node v_i where $Index_{v_i}$ denotes the index of the node, and $IsStart_v$ is a binary variable indicating whether v_i is the start node or not. Finally, the output is a pointer for each node $y_{v_i} \in V$ indicating the parent of the node after the execution of BFS.

3.1 Architecture

Our proposed method follows an encode-process-decode procedure[4]. Namely, we first encode all the features into an embedding space using a neural network: $h_{v_i} = f_{enc}(x_{v_i})$, then a processor P (which is another neural network) is applied recursively and is supposed to perform one step of message-passing: $h_{v_i}^{(t+1)} = P(h_{v_i}^{(t)}, h_{\mathcal{N}(v_i)}^{(t)}, A)$ where $h_{v_i}^{(t)}$ is the embedding of node v_i at step t. Finally, after several steps of intermediate processing, the hidden representations of $h_{v_i}^{(t)}$ would contain the output of the algorithm. So, the outputs are computed using a neural network decoder: $\hat{y} = f_{dec}(x_V)$. Previous methods also leverage the intermediate outputs of algorithms as a supervision, but, we focus on an end-to-end training. We keep f_{enc} and f_{dec} as simple MLPs and focus on improving the processor P.

3.1.1 Encoder

To encode the features of the given nodes, we concatenate all given node features (e.g., node index) and all graph features (e.g., the query for binary search). Further, we append random uniform features to the node features [2]. Therefore, the final node features would contain the following vector:

$$x'_{v_i} = [x_{v_i}, x_g, \epsilon_{v_i}] \quad \forall\, 1 \le i \le n,$$

where ϵ_{v_i} is a d_ϵ-dimensional vector with elements samples $i.i.d.$ from a uniform distribution $\mathcal{U}(-1, 1)$. Note that the random features are used as a data augmentation manner and are resampled every time a datapoint is fed into the model. Finally, the features are encoded into a high-dimensional space using a linear layer f_{enc}:

$$h_{v_i}^{(0)} = f_{enc}(x'_{v_i}) \quad \forall\, 1 \le i \le n.$$

3.1.2 Processor

The processor takes edge attributes and node embeddings at step t, performs one step of processing and updates the embeddings of nodes to get the embeddings of step $t+1$. As we do not use any intermediate supervision, we keep the number of processing steps r independent of the underlying algorithm. For simplicity, we keep it a large enough constant number.

The architecture of the processor is similar to a pre-norm transformer [9], except that it utilizes GNNs heads instead of self-attention heads. Namely, the next step embedding is computed using the following steps:

$$\begin{aligned}
z_1 &= LayerNorm(h_{v_i}^{(t)}) \\
z_2 &= Concatenate([GNN_1(z_1), GNN_2(z_1)]) \\
z_3 &= h_{v_i}^{(t)} + LayerNorm(z_2) \\
h_{v_i}^{(t+1)} &= LayerNorm(FC(z_3)) + z_3,
\end{aligned}$$

where $h_{v_i}^{(t)}, h_{v_i}^{(t+1)}, z_1, z_2, z_3 \in \mathbb{R}^{d_{model}}$. Moreover, GNN_1 and GNN_2 are two separate graph network blocks; GNN_1 operates on incoming nodes of a graph (i.e., source to target message passing) and GNN_2 operates on outgoing nodes (i.e., target to source message passing). This bidirectional design choice is to facilitate learning graph algorithms containing directed edges (e.g., DFS, DAG shortest path). For the GNN backbones, we use a general message passing neural network analogous to EdgeConv[8]:

$$m_{v_i,v_j} = FC([z_{v_i}, z_{v_j}, x_{e_{v_i,v_j}}]), \quad z_v^{'} = \max(\{m_{u,v_i} \,|\, u \in \mathcal{N}(v_i)\}) \quad \forall\, 1 \leq i, j \leq n,$$

where $x_{e_{v_i,v_j}}$ is the edge features of the edge between node v_i and v_j, FC is a two-layer fully connected neural network, and $\mathcal{N}(v_i)$ is the set of neighbors of v_i (i.e., incoming nodes for GNN_1 and outgoing nodes for GNN_2).

3.1.3 Decoder

After the execution of several steps of processing, the decoder f_{dec} takes the node embeddings and generates outputs of the underlying algorithmic task. Depending on the task, f_{dec} is a two-layer model.

4 Experiments

We mainly run our experiments on the CLRS benchmark. The benchmark contains 30 algorithmic reasoning tasks. For each task, the training set contains 1000 graphs/sequences of length 16 (e.g., graph with 16 nodes). The validation set contains 32 graphs/sequences from the same distribution. The test set contains 32 graphs/sequences of length 64 and same generation strategy as training. The goal is to measure the generalization of a model on larger inputs. The graphs are generated using Erdős–Rényi generation procedure, and all other inputs are generated uniformly at random for other parts (see Appendix A for more details).

Implementation Details. We set $d_{model} = 128$ as the embedding size of our architecture. We use Adam optimizer with learning rate 0.0001 reduced to 0.00001 using a cosine annealing scheduler. The gradients are clipped to a norm of 1.0. We fix the number of recursive steps to $r = 64$ steps. Training is done for $30,000$ optimization steps and batch-size of 128. We use early stopping on validation score and keep the model snapshot with highest validation accuracy. Since the experiments are costly, we only run our method for each algorithm only once and do not rerun it for several seeds.

Evaluation Metrics. Each algorithmic task has a different type of output, and hence different evaluation metric. For tasks with categorical/pointer outputs the metric is accuracy. For the rest of the tasks with binary mask outputs, the metric is f1 metric to account for imbalanced classes. Moreover, for models with more than one output, the score is the average of all outputs.

Table 1: Test scores of all 30 algorithms.

Method	Ours	CLRS PGN[6]	CLRS MPNN[6]
articulation points	**55.93**	54.79	50.06
activity selector	89.05	66.40	**92.98**
bellman ford	55.62	66.26	**66.60**
bfs	**94.73**	87.74	82.32
binary search	17.38	**25.00**	21.88
bridges	**64.41**	31.31	30.04
bubble sort	**77.29**	22.80	37.50
dag shortest paths	**85.55**	62.35	74.80
dfs	**20.65**	7.03	6.20
dijkstra	55.62	**76.86**	73.34
find maximum subarray kadane	**22.51**	17.19	21.88
floyd warshall	23.02	25.92	**28.17**
graham scan	90.31	56.44	**96.74**
heapsort	**77.29**	27.00	33.30
insertion sort	**77.29**	28.32	26.66
jarvis march	90.31	58.15	**95.18**
kmp matcher	**6.69**	3.12	3.12
lcs length	**85.84**	60.18	57.24
matrix chain order	**81.26**	79.41	79.33
minimum	**98.93**	90.62	3.12
mst kruskal	52.96	**72.51**	70.47
mst prim	20.65	47.31	**55.57**
naive string matcher	**7.08**	3.12	0.00
optimal bst	**74.93**	73.38	72.32
quickselect	**21.48**	9.38	6.25
quicksort	**77.29**	29.39	38.04
segments intersect	89.92	80.00	**91.43**
strongly connected components	**64.45**	50.73	61.82
task scheduling	82.31	81.82	**89.01**
topological sort	**77.69**	62.79	61.87
Mean	**61.28**	48.58	50.91

4.1 Comparison with CLRS benchmark

In this section, we evaluate our proposed method on the 30 tasks of the CLRS benchmark and compare with the baseline on all of them. Finally, we will report the mean accuracy on all tasks. The results are shown in Table 1. As the results suggest, we achieve the state-of-the-art results on tasks on OOD generalization

4.2 Infinite data regime

We run experiments on larger dataset to test if the in-distribution dataset is large enough or not.To this end, we generate a larger version of CLRS benchmark and call it *CLRS-large*. This dataset contains 10 times more train/validation/test data (i.e. $10,000$ training sequences, 320 validation sequences, and 320 test sequences). The results are shown in Table 2. Interestingly, the generalization of the majority of tasks improve significantly with availability of more in-distribution data. Hence, the CLRS dataset does not seem to contain enough datapoints for a model to generalize properly.

5 Conclusion

In this project, we proposed a graph neural networks-based architecture to execute algorithms using deep learning. Our method exhibits a 10.37%average improvement over the baseline on the CLRS

Table 2: Test scores of our method on CLRS and CLRS-large ($10X$ more data). While the score of some tasks are saturated, the majority of tasks could benefit from larger in-distribution dataset to significantly generalize out-of-distribution.

Dataset	CLRS	CLRS-large
articulation points	55.93	**94.47**
activity selector	89.05	**94.72**
bellman ford	55.62	**84.31**
bfs	94.73	**98.02**
binary search	17.38	**17.78**
bridges	64.41	**96.64**
bubble sort	77.29	**78.42**
dag shortest paths	85.55	**95.01**
dfs	**20.65**	20.03
dijkstra	55.62	**84.31**
find maximum subarray kadane	22.51	**26.62**
floyd warshall	**23.02**	19.20
graham scan	90.31	**97.34**
heapsort	77.29	**78.42**
insertion sort	77.29	**78.42**
jarvis march	90.31	**97.34**
kmp matcher	6.69	**20.13**
lcs length	85.84	**86.62**
matrix chain order	**81.26**	80.37
minimum	98.93	**99.22**
mst kruskal	52.96	**79.65**
mst prim	20.65	**52.22**
naive string matcher	7.08	**30.79**
optimal bst	**74.93**	72.79
quickselect	**21.48**	16.68
quicksort	77.29	**78.42**
segments intersect	89.92	**96.73**
strongly connected components	64.45	**68.11**
task scheduling	82.31	**83.61**
topological sort	77.69	**84.12**
Mean	61.28	**70.35**

benchmark. We further ran more experiments and showed that with more in-distribution datapoints, the model could benefit from the data and further generalize to OOD.

Future work may work on improving the expressivity of our proposed architecture to further improve the OOD generalization. Moreover, going beyond the message-passing bottleneck and using global information is interesting – enabling the architecture to generalize to any graph-size without the need to add more layers for larger sparser graphs. Finally, it would be interesting to apply our general-purpose framework to NP-Hard problems and evaluate its performance in finding an approximate solution those types of tasks.

References

[1] Gabriele Corso, Luca Cavalleri, D. Beaini, Pietro Lio', and Petar Velickovic. Principal neighbourhood aggregation for graph nets. *ArXiv*, abs/2004.05718, 2020.

[2] Ryoma Sato, Makoto Yamada, and Hisashi Kashima. Random features strengthen graph neural networks. In *Proceedings of the 2021 SIAM International Conference on Data Mining, SDM*, 2021.

[3] Ashish Vaswani, Noam M. Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *ArXiv*, abs/1706.03762,

2017.

[4] Petar Velickovic and Charles Blundell. Neural algorithmic reasoning. *Patterns*, 2, 2021.

[5] Petar Velickovic, Rex Ying, Matilde Padovano, Raia Hadsell, and Charles Blundell. Neural execution of graph algorithms. *ArXiv*, abs/1910.10593, 2020.

[6] Petar Veličković, Adrià Puigdomènech Badia, David Budden, Razvan Pascanu, Andrea Banino, Misha Dashevskiy, Raia Hadsell, and Charles Blundell. The clrs algorithmic reasoning benchmark. 2021.

[7] Petar Velivckovi'c, Lars Buesing, Matthew Overlan, Razvan Pascanu, Oriol Vinyals, and Charles Blundell. Pointer graph networks. *ArXiv*, abs/2006.06380, 2020.

[8] Yue Wang, Yongbin Sun, Ziwei Liu, Sanjay E. Sarma, Michael M. Bronstein, and Justin M. Solomon. Dynamic graph cnn for learning on point clouds. *ACM Transactions on Graphics (TOG)*, 2019.

[9] Ruibin Xiong, Yunchang Yang, Di He, Kai Zheng, Shuxin Zheng, Chen Xing, Huishuai Zhang, Yanyan Lan, Liwei Wang, and Tie-Yan Liu. On layer normalization in the transformer architecture. *ArXiv*, abs/2002.04745, 2020.

[10] Keyulu Xu, Jingling Li, Mozhi Zhang, Simon S. Du, Ken ichi Kawarabayashi, and Stefanie Jegelka. What can neural networks reason about? In *International Conference on Learning Representations*, 2020.

[11] Yujun Yan, Kevin Swersky, Danai Koutra, Parthasarathy Ranganathan, and Milad Hashemi. Neural execution engines: Learning to execute subroutines. *ArXiv*, abs/2006.08084, 2020.

[12] Lingxiao Zhao and Leman Akoglu. Pairnorm: Tackling oversmoothing in gnns. In *International Conference on Learning Representations*, 2020.

A Dataset Details

Here, we provide more details of the CLRS benchmark. The benchmark contains the following 30 algorithms: articulation points, activity selector, bellman ford, bfs, binary search, bridges, bubble sort, dag shortest paths, dfs, dijkstra, find maximum subarray kadane, floyd warshall, graham scan, heapsort, insertion sort, jarvis march, kmp matcher, lcs length, matrix chain order, minimum, mst kruskal, mst prim, naive string matcher, optimal bst, quickselect, quicksort, segments intersect, strongly connected components, task scheduling, topological sort.

Each tasks contains some inputs, and outputs. The benchmark also contains information about the intermediate steps of each algorithm (calling them hints), which we do not use or discuss in our work. Each input/output may be attributed to the following locations:

- Node: nodes attributes. For instance, node index, scalar value of an element in an array.
- Edge: edge attributes such as edge weights, edge mask (i.e., adjacency matrix).
- Graph: node/edge independent variables such as a query value for binary search tree.

Moreover, each input/output gets one of the following types:

- Scalar: scalar values. For example, values stored in an array, edge weights.
- Categorical: categorical values such as color of each node.
- Pointer: pointers to other entities. For instance, a pointer for each node in BFS and DFS algorithms.
- Mask: binary masks on different entities such as adjacency matrix of a graph on edges or output of a string matching algorithm on nodes.

Table 3: Test scores of our method on CLRS and CLRS-large with and without using random features.

Dataset Random Features	CLRS No	CLRS Yes	CLRS-large No	CLRS-large Yes
articulation points	50.40	55.93	65.14	94.47
activity selector	75.77	89.05	91.72	94.72
bellman ford	48.05	55.62	79.03	84.31
bfs	82.03	94.73	93.97	98.02
binary search	23.24	17.38	24.53	17.78
bridges	43.47	64.41	81.06	96.64
bubble sort	43.60	77.29	62.74	78.42
dag shortest paths	81.20	85.55	94.34	95.01
dfs	11.23	20.65	21.02	20.03
dijkstra	48.05	55.62	79.03	84.31
find maximum subarray kadane	12.26	22.51	26.82	26.62
floyd warshall	13.43	23.02	11.81	19.20
graham scan	79.00	90.31	96.54	97.34
heapsort	43.60	77.29	62.74	78.42
insertion sort	43.60	77.29	62.74	78.42
jarvis march	79.00	90.31	96.54	97.34
kmp matcher	5.96	6.69	8.06	20.13
lcs length	86.39	85.84	85.40	86.62
matrix chain order	76.46	81.26	80.42	80.37
minimum	97.02	98.93	99.85	99.22
mst kruskal	29.54	52.96	66.97	79.65
mst prim	17.19	20.65	44.76	52.22
naive string matcher	6.01	7.08	8.42	30.79
optimal bst	73.08	74.93	73.62	72.79
quickselect	8.69	21.48	8.33	16.68
quicksort	43.60	77.29	62.74	78.42
segments intersect	88.57	89.92	96.50	96.73
strongly connected components	55.66	64.45	65.83	68.11
task scheduling	79.19	82.31	71.78	83.61
topological sort	76.88	77.69	84.88	84.12
Mean	50.74	61.28	63.58	70.35

B Ablation Studies

B.1 Random Features

To measure the effect of random features, we run experiment with and without the random features on both CLRS dataset and CLRS-large dataset. the results are shown in Table 3, random features play a huge role in the generalization of the model. However, their effect diminishes as the dataset size gets larger. Hence, we speculate that it acts as a data-augmentation and regularization technique.

B.2 Processor Block

We try out different processor types as the GNN_1 and GNN_2 backbones. The baselines are as follows:

- EdgesConv: The processor that we use in our method, with maximum aggregation.
- PNA[1]: This processor contains three aggregations, maximum, minimum, mean.
- Transformer: A transformer attention style graph neural network. On a fully connected graph, the architecture would become exactly a pre-norm transformer[9].
- Top5 Trasnformer: A transformer self-attention variant which only keeps the attention of top 5 neighbors. This is to enforce attention sparsity on the model.

Table 4: Test scores of using different processors on CLRS dataset. Symbol "-" means that particular experiment exceeded the time/memory limit.

Method	EdgeConv	PNA	Temp Trasnformer	Top5 Trasnformer	Transformer
articulation points	**55.93**	55.09	48.25	50.21	48.77
activity selector	**89.05**	59.35	83.64	77.51	74.50
bellman ford	55.62	43.99	**79.10**	78.56	-
bfs	94.73	92.43	**97.75**	97.07	96.88
binary search	17.38	16.80	17.43	15.58	**19.92**
bridges	64.41	**78.78**	39.57	49.28	53.34
bubble sort	77.29	79.74	**84.13**	74.46	11.52
dag shortest paths	85.55	78.61	**95.41**	93.85	93.16
dfs	**20.65**	8.69	12.21	20.41	19.14
dijkstra	55.62	48.49	76.03	74.17	**78.37**
find maximum subarray kadane	22.51	16.11	22.46	24.27	**36.87**
floyd warshall	**23.02**	11.74	16.35	21.69	14.94
graham scan	90.31	80.70	**95.70**	95.42	91.88
heapsort	77.29	**84.96**	73.58	52.15	12.30
insertion sort	77.29	71.58	62.50	**77.59**	10.60
jarvis march	90.31	78.81	93.87	**95.03**	92.15
kmp matcher	6.69	5.18	5.96	**7.08**	-
lcs length	**85.84**	85.10	85.72	80.99	-
matrix chain order	81.26	76.49	80.26	78.32	**81.45**
minimum	**98.93**	97.27	98.14	97.36	97.51
mst kruskal	52.96	64.66	**77.37**	67.23	-
mst prim	20.65	16.50	**36.33**	30.62	22.51
naive string matcher	7.08	3.76	7.03	6.88	**7.13**
optimal bst	**74.93**	69.61	73.07	65.61	70.76
quickselect	**21.48**	2.34	12.45	6.25	2.78
quicksort	77.29	52.64	**83.20**	74.32	8.84
segments intersect	89.92	**92.06**	88.79	85.46	82.27
strongly connected components	**64.45**	60.25	63.96	60.60	-
task scheduling	82.31	**85.07**	84.90	81.84	-
topological sort	77.69	75.39	80.15	80.47	**81.57**
Mean	61.28	56.41	**62.51**	60.68	-

- Temp Trasnformer: Another way of imposing attention sparsity to tansformer is to decrease the temperature of its softmax. This way, the attention coefficients become sharper.

The results are present in Table 4. We observe that no processor dominate the others and each one performs well in some subsets of tasks.