Beyond Erdős-Rényi: Generalization in Algorithmic Reasoning

Dobrik Georgiev
University of Cambridge
dgg30@cam.ac.uk

Pietro Liò
University of Cambridge
pl219@cam.ac.uk

Jakub Bachurski
University of Cambridge
jkb55@cam.ac.uk

Junhua Chen
University of Cambridge
jc2318@cam.ac.uk

Tunan Shi
University of Cambridge
ts862@cam.ac.uk

Lorenzo Giusti
Università La Sapienza, Roma
giusti@diag.uniroma1.it

Abstract

Neural algorithmic reasoning excels in many graph algorithms, but assessment mainly focuses on the Erdős-Rényi (ER) graph family. This study delves into graph algorithmic models' generalization across diverse distributions. Testing a leading model exposes overreliance on ER graphs for generalization assessment. We further investigate two scenarios: generalisation to every target distribution and single target distributions. Our results suggest that achieving the former is not as trivial and achieving the latter can be aided by selecting source distribution via novel Tree Mover's Distance interpretation.

1 Introduction

When implementing algorithms for real-world problems, a crucial decision is how to appropriately represent target problem elements and input data, often complicated due to diverse types like sequences or images. Classical algorithms can lead to a bottleneck [13] by compressing data into a single scalar, while Neural Algorithmic Reasoning (NAR) aims to overcome this by using neural models to simulate algorithm execution in a high-dimensional latent space, finding applications in various fields [13, 5, 2, 11]. However, NAR lacks correctness guarantees and, though progress shows promise in generalization and accuracy, it has been tested on mainly on the Erdős-Rényi [6] graph type.

In our paper, we set out to investigate if it is worth including other families of random graphs in the analysis of NAR performance on graph algorithms and also training of NAR models. Concretely, our contributions are as follows:

- We illustrate the usefulness of other graph families for generating diverse test cases, as 5 out of 12 algorithms we examined performed significantly worse on additional graph families.
- Moreover, we find that data augmentation by adding graphs from all tested families to the training data does not notably enhance performance across all domains. In some cases, such augmentation adds noise and hampers training success.
- When targeting a specific graph distribution, training on a close distribution gives best results. We define closeness by adapting the Tree Mover's Distance [3] and correlate it with the generalisation accuracy. To the best of our knowledge we are the first to analyse generalisation of algorithmic reasoners using any graph theoretical metrics.

2 Background

Neural algorithmic reasoning (NAR). To advance NAR, the CLRS-30 benchmark was introduced [12], encompassing 30 algorithms. While not all are graph-based, the input to every task is a graph that may have a set of node, edge or graph input features. Appendix A provides a brief CLRS-30

Georgiev et al., Beyond Erdős-Rényi: Generalization in Algorithmic Reasoning (Extended Abstract). Presented at the Second Learning on Graphs Conference (LoG 2023), Virtual Event, November 27–30, 2023.

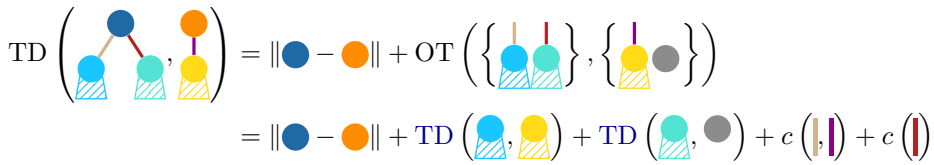


Figure 1: Edge-featured computational **T**ree **D**istance, the foundation of our TMD variant. The gray node signifies a blank node [cf. 3], while edge features impact optimal transport distance. c denotes the cost for edge features coming from computing the optimal transport (OT). Subtrees of a node are denoted with a hatch pattern. Recursion is highlighted in blue.

overview. As non-graph algorithms like sorting have fixed input graph structures (only size and features vary), we focus on graph algorithms within the benchmark. Notably, in CLRS-30, the best performing open-source model, Triplet-MPNN, an MPNN-based neural network, operates on complete graphs with added binary edge features indicating edge presence, similar to the original MPNNs [7].

To simulate an algorithm $\mathcal{A}$ we structure our model by the encode-process-decode architecture [8]. The encoder network f consists of a linear transformation for each input and hint. A GNN is used for the processor p that serves to mimick $\mathcal{A}$'s steps. The decoder g, similar to f is also a linear transformation. (More details on encode-process-decode can be found in Appendix B.) In NAR, robust out-of-distribution (OOD) generalization is vital. While training on "small" graphs up to 16 nodes, testing occurs on $4\times$ larger graphs. This tests whether the model grasps algorithm behavior and can extrapolate effectively. Evaluating on OOD data gauges generalization, ensuring reliability across diverse input sizes. This strategy is inspired by classical algorithms' size-invariance property, maintaining solution correctness regardless of input scale.

Optimal transport and Tree Mover's Distance. The Tree Mover's distance (TMD) [3] is a psuedometric on attributed graphs. It takes into account both the computational tree structure of the graph and the attributes of the nodes. Formally, for two graphs G_A, G_B of size N the *computation trees* $A_1 \dots A_N$ and $B_1 \dots B_N$ for both graphs are generated up to a fixed depth L. Subsequently, a distance value d_{ij}, defined via hierarchical optimal transport, is computed for all pairs of computational trees A_i and B_j. The Tree Mover's distance between A and B is then defined as the result of solving the assignment problem on this tableau of distances.

The metric helps capture the extent to which the local structures of the graphs being compared differs, with higher Tree Mover's distances corresponding to more dissimilar graphs. The original paper uncovered a strong relation between the the performance of a model on a test dataset and the Tree Mover's distance between the graphs of the test dataset and the training set.

3 How and why do our algorithmic reasoners generalise?

In this section we will try to answer how algorithmic reasoners generalise when deployed on *any* data distribution, similar to their deterministic counterparts, and when deployed on a *target* data distribution. To aid our analysis, we will first introduce our adaptation of TMD.

3.1 Normalised Tree Mover's Divergence with Edge Features

Figure 1 depicts our Tree Distance (TD, the building block of TMD) with edge features. The core insight is to factor in edge features from root to next level node vectors when calculating optimal transport between computational subtree sets and exclude them at next steps of the recursion. Due to space limits, more rigorous formulation is given in Appendix C.

The next change to the original TMD is based on the hypothesis that families with high pairwise distance between randomly chosen graphs likely exhibit diverse structures, potentially improving generalization when trained on them. Taking this and the existing hypothesis [3] into account – that models trained on one graph family tend to generalize better to families with low Tree Mover's distance to the training family – we define the *normalized Tree Mover's divergence* between a training graph distribution A and a testing graph distribution B:

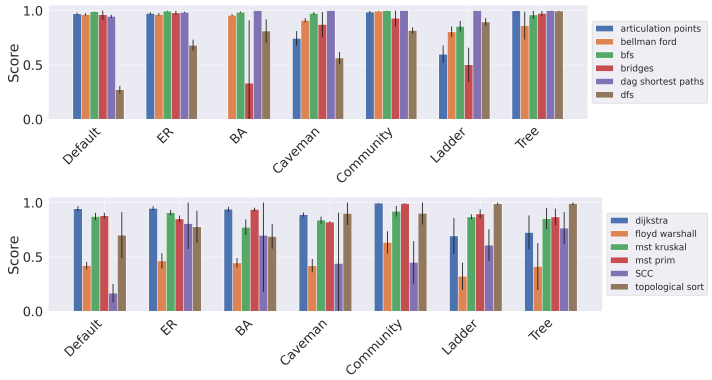


Figure 2: Generalisation of algorithmic reasoning model to different graph families. The set of algorithms is split in two figures to improve readability.

$$\mathrm{NTMD}(A, B) = \frac{\mathbb{E}_{G_1 \sim A, G_2 \sim B}[\mathrm{TMD}(G_1, G_2)]}{\mathbb{E}_{G_1 \sim A, G_2 \sim A}[\mathrm{TMD}(G_1, G_2)]}$$

We would also like to note that our proposed metric can also be viewed as an improvement on TMD by 1) factoring in edge features and 2) making it an asymmetric measure (hence the use of "divergence"). One might expect that in many circumstances training on distribution A and testing on distribution B will yield very different performances compared to training on distribution B and testing on distribution A, but this is not accounted for by the original TMD.

3.2 Experimental setup

Our experiments are implemented on top of CLRS-30 (3 different random seeds/experiment).

Data generation. For our initial experiments our training data distribution remains the same as in CLRS-30: Erdős-Rényi family with edge probablity p varied between 0.1 and 0.9. When testing we always keep the size distribution shift (increasing the number of nodes by $4\times$), but we also consider varying the types of graphs during test time. Concretely, we take inspiration from Velickovic et al. [14] and consider 32 test graphs (as in CLRS-30) from the following set of families: **Erdős-Rényi** (ER), **Trees**, **Barabási-Albert**, **4-Caveman graphs**, **4-Community graphs**, **Ladder graph**, **Default**. The last "family" is the pre-generated test set that CLRS-30 provides (never used for training). For futher details, refer to Appendix D.

To handle algorithms operating on weighted graphs, we utilized the same weight distribution as in CLRS-30. This decision was primarily motivated by the fact that neural networks often face challenges in generalizing values, even struggling with simple tasks like arithmetic addition [10].

Training. For our base experiment we train using the default settings provided on the CLRS-30 open source code. They correspond to the best configuration in Ibarz et al. [9]. In our ablations, we extend the training procedure by incorporating samples from non-Erdős-Rényi (non-ER) distributions while keeping other aspects constant. When constructing a training batch, if more than one family is available, we uniformly select a single family from which to sample.

3.3 Results

One family is not enough. We present our first observation in figure Figure 2. When compared to ER test set, for 5/12 algorithms there was at least one family of graphs where the performance deteriorated significantly. Results also extend to the default test set provided, and there were again 5 algorithms which exhibited significantly worse performance on a different test set.[1] Based on the above observations, we highly encourage future works in algorithmic reasoning to consider evaluating their models on graph distributions beyond ER.

[1]For the exact lists of algorithms and what do we mean by "significantly worse performance", see Appendix E.

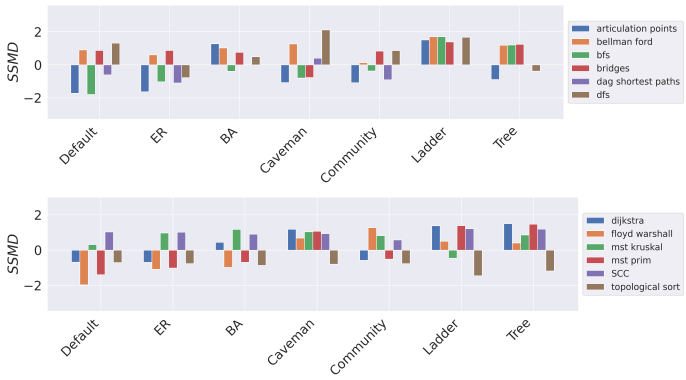


Figure 3: Adding richer data often does not produce significantly better performance.

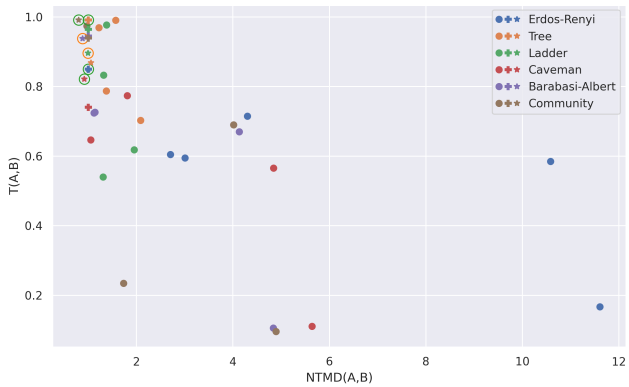


Figure 4: Plot of NTMD against T for Prim's algorithm. Correlation coefficient $r = -0.636$ ($\rho = 5.9 \times 10^{-6}$). Each symbol represents a source dataset, each color – target dataset. **Plus** is used if source matches target, **star** – if source is ER, **green/orange outside** – if picking the smallest TMD gives/does not give highest performance on target dataset.

Challenges Encountered When Diversifying Data. We further explored the effect of diversifying our data. All models were trained using data from other classes, and we computed the strictly standardised mean difference (SSMD) [17] before (μ_1) and after the data augmentation (μ_2). Positive SSMDs imply that data augmentation improves performance (and vice versa). Small absolute values imply no statistical difference:

$$\text{SSMD} = (\mu_2 - \mu_1)/\sqrt{\frac{\sigma_1^2 + \sigma_2^2}{2}} \tag{1}$$

SSMDs are presented in Figure 3. For all algorithms there was no strong improvement (SSMD>2) across all datasets, except for DFS on Caveman graph family. Appendix Appendix H shows that these results extend to three other architectues.

Picking lowest NTMD often gives good performance. Results for Prim's algorithm are visualised in Figure 4 (remaining are in Appendix G; NTMD calculation details are in Appendix F). In all cases there was significant ($\rho < 0.05$) negative correlation between NTMD and T. Furthermore, in 16 out of 24 cases (4 algorithms, 6 target datasets), picking the lowest NTMD gives best performance. In the remaining 6 cases, performance was not far from the optimal one. If TMD is used instead no significant correlation could be found (Figure 7, Appendix G).

A final interesting observation is that often the ER distribution was quite close to the target distribution. It was, in fact, even the closest target distribution in 10/16 cases, further supporting our results that training on ER alone gives overall good performance.

References

[1] Barabási, A.-L. and Albert, R. (1999). Emergence of scaling in random networks. *science*, 286(5439):509–512. 7

[2] Beurer-Kellner, L., Vechev, M. T., Vanbever, L., and Velickovic, P. (2022). Learning to configure computer networks with neural algorithmic reasoning. In *NeurIPS*. 1

[3] Chuang, C.-Y. and Jegelka, S. (2022). Tree mover's distance: Bridging graph metrics and stability of graph neural networks. *Advances in Neural Information Processing Systems*, 35. 1, 2, 6, 7

[4] Cormen, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2009). *Introduction to Algorithms, 3rd Edition*. MIT Press. 6

[5] Deac, A., Veličković, P., Milinković, O., Bacon, P.-L., Tang, J., and Nikolić, M. (2020). Xlvin: executed latent value iteration nets. In *Learning Meets Combinatorial Algorithms at NeurIPS2020*. 1

[6] Erdős, P., Rényi, A., et al. (1960). On the evolution of random graphs. *Publ. Math. Inst. Hung. Acad. Sci*, 5(1):17–60. 1, 7

[7] Gilmer, J., Schoenholz, S. S., Riley, P. F., Vinyals, O., and Dahl, G. E. (2017). Neural message passing for quantum chemistry. In *International conference on machine learning*, pages 1263–1272. PMLR. 2

[8] Hamrick, J. B., Allen, K. R., Bapst, V., Zhu, T., McKee, K. R., Tenenbaum, J. B., and Battaglia, P. W. (2018). Relational inductive bias for physical construction in humans and machines. *arXiv preprint arXiv:1806.01203*. 2

[9] Ibarz, B., Kurin, V., Papamakarios, G., Nikiforou, K., Bennani, M., Csordás, R., Dudzik, A. J., Bosnjak, M., Vitvitskyi, A., Rubanova, Y., Deac, A., Bevilacqua, B., Ganin, Y., Blundell, C., and Velickovic, P. (2022). A generalist neural algorithmic learner. In Rieck, B. and Pascanu, R., editors, *Learning on Graphs Conference, LoG 2022, 9-12 December 2022, Virtual Event*, volume 198 of *Proceedings of Machine Learning Research*, page 2. PMLR. 3

[10] Klindt, D. A. (2023). Controlling neural network smoothness for neural algorithmic reasoning. *Transactions on Machine Learning Research*. 3

[11] Numeroso, D., Bacciu, D., and Veličković, P. (2023). Dual algorithmic reasoning. In *ICLR*. OpenReview.net. 1

[12] Veličković, P., Badia, A. P., Budden, D., Pascanu, R., Banino, A., Dashevskiy, M., Hadsell, R., and Blundell, C. (2022). The clrs algorithmic reasoning benchmark. In *International Conference on Machine Learning*, pages 22084–22102. PMLR. 1, 6, 7

[13] Velickovic, P. and Blundell, C. (2021). Neural algorithmic reasoning. *Patterns*, 2(7):100273. 1

[14] Velickovic, P., Ying, R., Padovano, M., Hadsell, R., and Blundell, C. (2020). Neural execution of graph algorithms. In *ICLR*. OpenReview.net. 3

[15] Wang, X., Wang, L., Wu, Y., et al. (2009). An optimal algorithm for prufer codes. *J. Softw. Eng. Appl.*, 2(2):111–115. 7

[16] Watts, D. J. (1999). Networks, dynamics, and the small-world phenomenon. *American Journal of sociology*, 105(2):493–527. 7

[17] Zhang, X. D. (2007). A new method with flexible and balanced control of false negatives and false positives for hit selection in rna interference high-throughput screening assays. *SLAS Discovery*, 12(5):645–655. 4

A CLRS-30

The *CLRS-30 benchmark* [12], or CLRS-30, comprises 30 iconic algorithms from the *Introduction to Algorithms* textbook [4]. The benchmark covers diverse algorithm types, including string algorithms, searching, dynamic programming, and graph algorithms. *All* data instances in the CLRS-30 benchmark are represented as graphs and are annotated with *input*, *output* and *hint* features and an associated position. Denote the dimensionality of a feature as F and $|V|$ as N. Input/output node features have the shape $N \times F$, edge features – $N \times N \times F$, graph features – only F. Hints encapsulate time series data of algorithm states. Like inputs/outputs, they include a temporal dimension which also indicates the duration of execution. All features fall into 5 types: `scalar`, `categorical`, `mask` (0/1 value), `mask_one` (only a single position can be 1), and `pointer`. Each feature type has an associated loss to be used when training the neural network [cf. 12].

B Neurally executing algorithms

The task-based encoder (and decoder) consists of a set of linear layers. At each timestep t, for a graph of n nodes and hidden dimensionality of h the task-based encoder f encodes each input and hint at every position into h-dimensional vector. f uses a separate linear layer for different inputs and hints, but sharing the weights across timesteps. Embeddings of inputs and hints located in the nodes/edges/graph all have the same dimension and are added together. At step t this gives node, edge and graph embeddings $\boldsymbol{x}_i^{(t)}, \boldsymbol{e}_{ij}^{(t)}, \boldsymbol{g}^{(t)}$ of shape $n \times h$, $n \times h \times h$, h.

The embeddings are fed through a MPNN processor P with latent state (similar to RNNs).

$$\boldsymbol{z}_i^{(t)} = \boldsymbol{x}_i^{(t)} \| \boldsymbol{h}_i^{(t-1)} \qquad \boldsymbol{m}_i^{(t)} = \max_{1 \le j \le n} f_{msg}(\boldsymbol{z}_i^{(t)}, \boldsymbol{z}_j^{(t)}, \boldsymbol{e}_{ij}^{(t)}, \boldsymbol{g}^{(t)}) \qquad \boldsymbol{h}_i^{(t)} = f_r(\boldsymbol{z}_i^{(t)}, \boldsymbol{m}_i^{(t)}) \quad (2)$$

where $\|$ denotes concatenation, f_{msg} is the GNN message function, f_r the GNN readout function and $\boldsymbol{h}^{(0)} = \mathbf{0}$.

The decoder g predicts hints for *next* step as well as final outputs at the final step. Again, it is parametrised as linear layers for each hint and output. Perhaps the only more specific type to decode is the `pointer` type where a similarity s_{ij} is computed for each node pair which is then normalised via $\mathrm{softmax}_j s_{ij}$ to obtain pointer probabilities. ($\arg\max$ is used if when obtaining actual pointer for, e.g., accuracy calculation)

C TMD with edge features

We use the notation from Chuang and Jegelka [3]. Let $T = (V, E, r)$ denote a rooted tree. We also let $\mathcal{T}_v$ be the multiset of computation trees of the node v which consists of trees that root at the descendants of v. Additionally, we define a blank tree $T_{\mathbb{0}}$ as a tree that contains a single node and no edges, and where the node feature is the zero vector $\mathbb{0}_p \in \mathbb{R}^p$. $T_{\mathbb{0}}^n$ is used to denote a multiset of n blank trees. We denote $\mathcal{T}_v^L := \{T_u^{L-1}\}_{u \in \mathcal{N}(v)}$ as the multiset of computation trees at node v which root at the descendants of v.

Definition C.1 (Edge Augmented Computation Trees)**.** Given a vertex v we define its multiset of edge augmented computation trees $\mathcal{T}_v'^L := \{(T_u^{L-1}, e_{vu})\}_{u \in \mathcal{N}(v)}$. It consists of all elements of $\mathcal{T}_v^L$, but with the edge from v attached to them as in Figure 1.[2]

Definition C.2 (Blank Tree Augmentation)**.** Given two multisets of trees $\mathcal{T}_u'^L$, $\mathcal{T}_v'^L$, ρ is defined as:

$$\rho : (\mathcal{T}_v', \mathcal{T}_u') \mapsto \left(\mathcal{T}_v' \bigcup \left(T_{\mathbb{0}}^{\max(|\mathcal{T}_u| - |\mathcal{T}_v|, 0)}, e_0\right), \mathcal{T}_u' \bigcup \left(T_{\mathbb{0}}^{\max(|\mathcal{T}_v| - |\mathcal{T}_u|, 0)}, e_0\right)\right) \quad (3)$$

Recalling that $\mathrm{OT}_d(X, Y)$ denotes the unnormalized version of the earth mover's distance, where d is the distance function between pairs in X and Y, we suggest the following *tree* distance function:

$$\mathrm{TDA}_w(T_a, T_b) := \begin{cases} \|x_{r_a} - x_{r_b}\| + w(L) \cdot \mathrm{OT}_{\mathrm{TD}'_w}\left(\rho\left(\mathcal{T}'_{r_a}, \mathcal{T}'_{r_b}\right)\right) & \text{if } L > 1, \\ \|x_{r_a} - x_{r_b}\| & \text{otherwise} \end{cases} \quad (4)$$

[2] Since the edge does not connect to a node, we represent this construction as a tuple (tree, edge).

The changed parts of the equation have been highlighted in red. The TD'_w distance function is defined as:

$$\text{TD}'_w((T_a, e_{aa'}), (T_b, e_{bb'})) := \|e_{aa'} - e_{bb'}\| + \text{TDA}_w(T_a, T_b) \tag{5}$$

The *graph* TMD distance remains unchanged from Chuang and Jegelka [3], but we use TDA_w instead of TD_w as a tree distance metric.

D Data generation

Test families were generated as follows:

- **Erdős-Rényi** (ER) [6] – these are same graphs as our training data, but with increased size.
- **Trees** – The trees are generated randomly from the Prüfer sequence [15].
- **Barabási-Albert** [1] – In the Barabási-Albert preferential attachment model we attach 2 or 3 nodes to every incoming node when the connectivity is halved and 4 or 5 otherwise.
- **4-Caveman graphs** [16] – Each clique is interconnected with probability p chosen uniformly from the set $\mathcal{P} = \{0.37, 0.44, 0.51, 0.58, 0.65, 0.72, 0.79, 0.86, 0.93\}$. p was subsequently halved if connectivity was halved in the original train dataset of CLRS-30.
- **4-Community graphs** – Each of the 4 communities is a ER subgraph with probability p drawn uniformly from $\mathcal{P} = \{0.55, 0.6, 0.7, 0.75, 0.65, 0.8, 0.85, 0.9, 0.95\}$. p was halved if the original connectivity was halved. Nodes between communities are then interconnected with $p = 0.01$.
- **Ladder graph** – To generate a ladder graph, we first generate a grid of size $N\prime = 2 \times \lceil N/2 \rceil$. To make $N\prime = N$ we discard one node at random if N is odd. As this may often generate very similar graphs and training instances, we randomly remove an edge with dropout probability 0.01.
- **Default** – Although not a family on its own, for better comparison we the also test on the pre-generated test set provided by Veličković et al. [12]. According to the paper authors, the graphs are "[...] Erdős-Rényi graphs with a certain edge probability" (p.8, [12]).

Some of our algorithms are defined on a directed graph. Our generators therefore return directed graphs, which we symmetrise only if the algorithm runs on undirected graphs. In the case of Ladder graphs, where this is trivial to achieve, we consider each undirected edge as two bi-directional edges and apply the edge dropout discussed above separately to each edge.

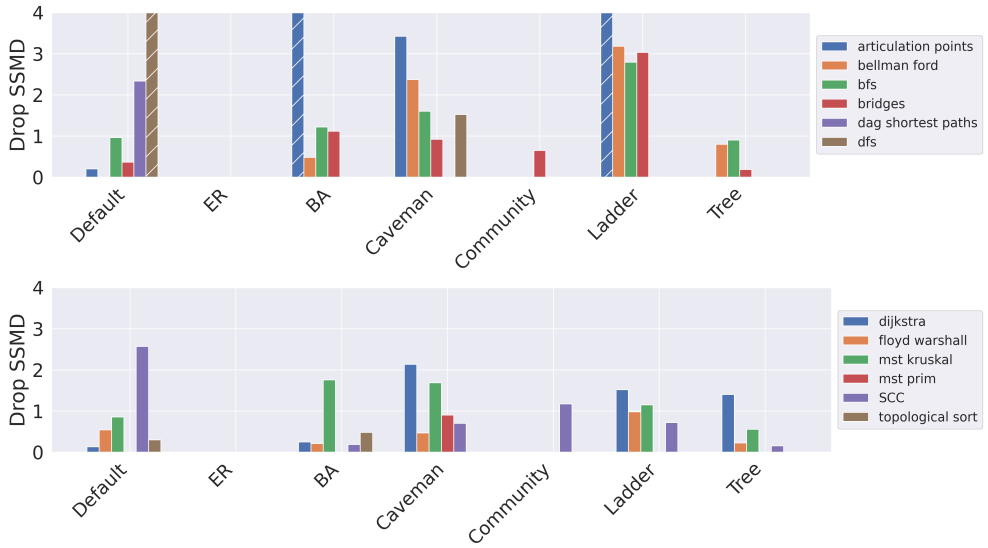
E Is test set result drop significant?

To easily obtain which are the algorithms with significant drop in performance on some dataset, we consider the Drop SSMD:

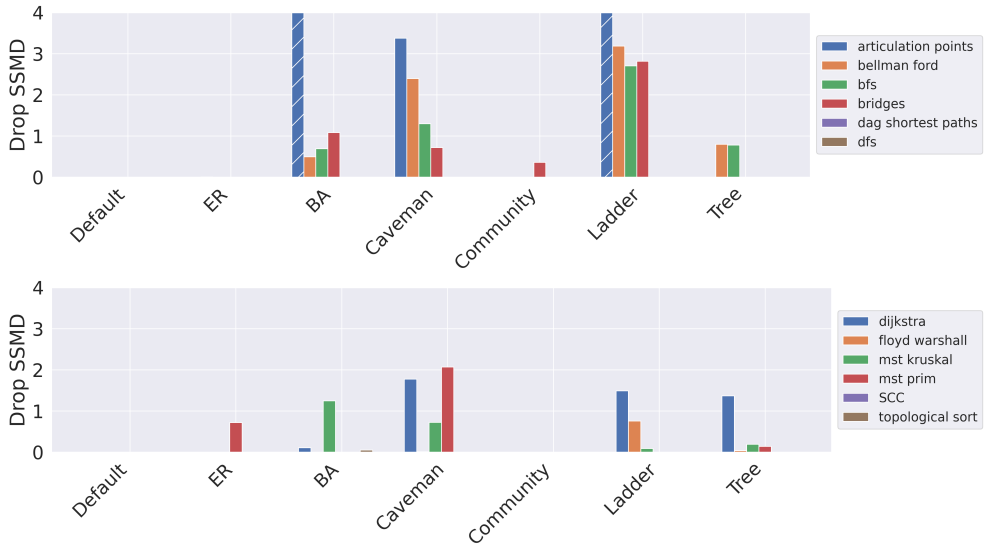
$$\text{Drop SSMD} = \frac{\max(\mu_1 - \mu_2, 0)}{\sqrt{\frac{\sigma_1^2 + \sigma_2^2}{2}}} \tag{6}$$

This metric ignores improvements in the score and allows us only to focus on datasets where accuracy worsened. We consider a drop significant, if it is ≥ 2. We then calculated for each algorithm the Drop SSMD w.r.t. the Erdős-Rényi dataset ($\mu_1 = \mu_{ER}$, measures how reliable using Erdős-Rényi is) and w.r.t. the Default test ($\mu_1 = \mu_{default}$, measures how reliable using Default is) set. Note that this metric requires a single training and only measures if using the additional test datasets capture significantly worse performances w.r.t. to the source dataset.

Results are visualised in Figure 5, all models are trained on Erdős-Rényi graphs. Testing on Erdős-Rényi alone, is not able to capture significant performance drops of: articulation points, Bellman-Ford, Breadth-First Search, Bridges and Dijkstra. Testing on the Default dataset alone: articulation points, Bellman-Ford, BFS, Bridges, MST-Prim.



(a) W.r.t. Erdős-Rényi



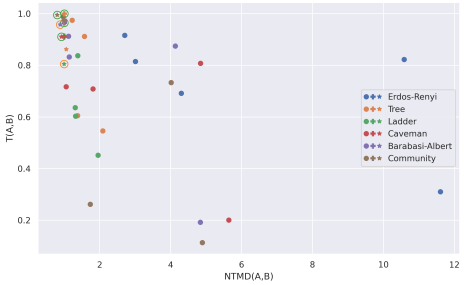
(b) W.r.t. Default

Figure 5: Drop SSMD. Bars with a hatch pattern, represent values larger than 4.

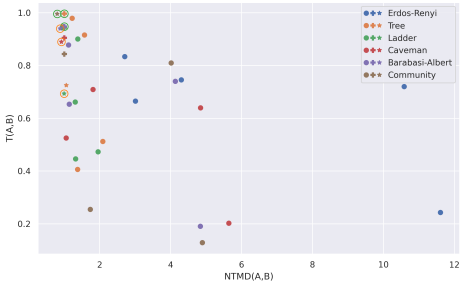
F NTMD calculation details

We compute $\mathrm{NTMD}(A, B)$, we use direct Monte Carlo sampling, averaging across 200 distinct graph pairs. TMD calculation employs $L = 4$ due to the computation expense of OT. Let the accuracy of testing on dataset B after training on A be $\mathrm{T}(A, B)$. We analyse the relationship between $\mathrm{NTMD}(A, B)$ and $\mathrm{T}(A, B)$ by training our model on 4 algorithms. We limited ourselves to such a low number, as obtaining results for a single one, requires 18 runs (3 seeds per 6 training families, 72 runs total for all 4 algorithms). The set of algorithms is: Bellman-Ford, Dijkstra, Floyd-Warshall, MST-Prim.

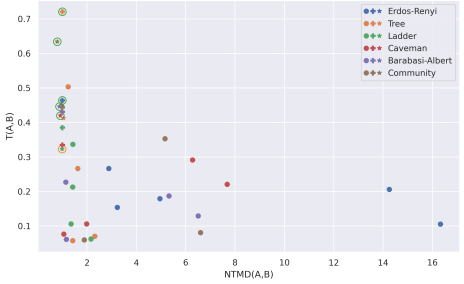
G Correlation plots of TMD and NTMD



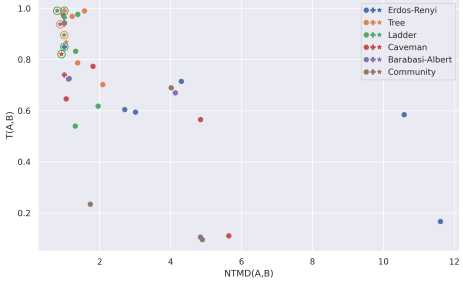
(a) Bellman-Ford $r = -0.470$ ($\rho = 0.002$).



(b) Dijkstra $r = -0.466$ ($\rho = 0.002$).

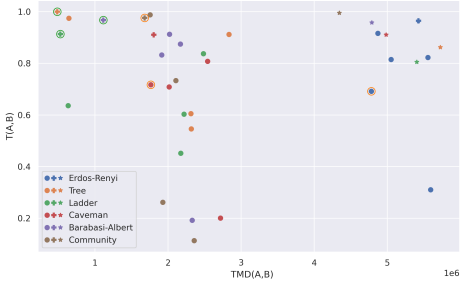


(c) Floyd-Warshall $r = -0.310$ ($\rho = 0.046$).

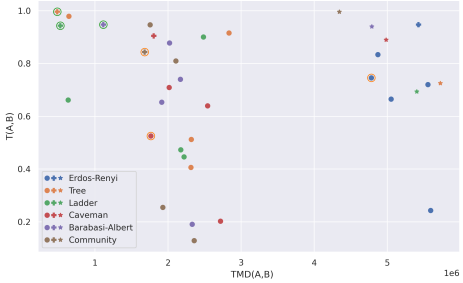


(d) Prim's $r = -0.636$ ($\rho = 5.9 \times 10^{-6}$).

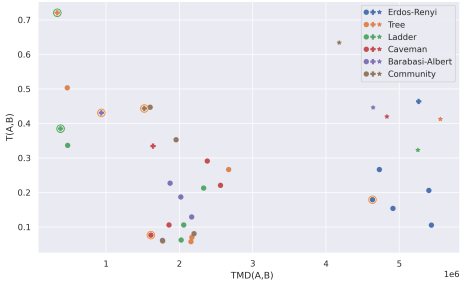
Figure 6: NTMD against T plots



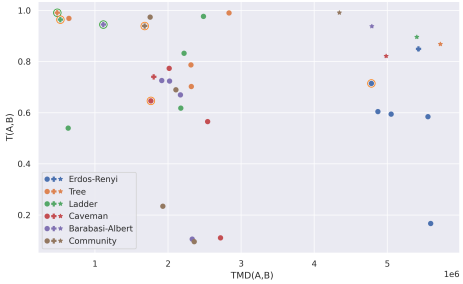
(a) Bellman-Ford $r = -0.013$ ($\rho = 0.927$).



(b) Dijkstra $r = -0.012$ ($\rho = 0.940$).



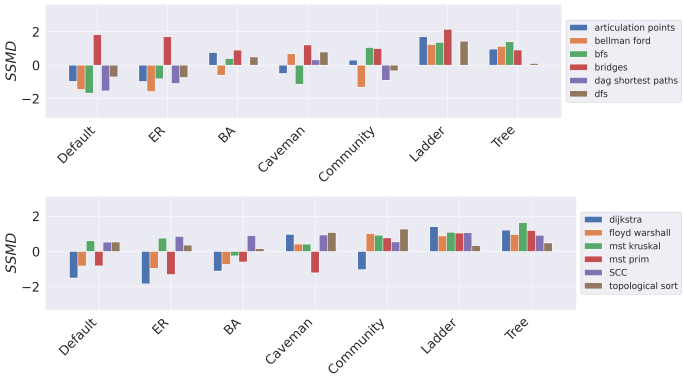
(c) Floyd-Warshall $r = -0.022$ ($\rho = 0.891$).



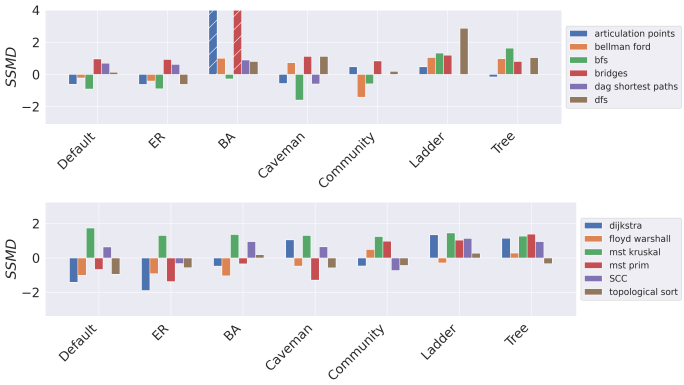
(d) Prim's $r = -0.052$ ($\rho = 0.743$).

Figure 7: TMD against T plots

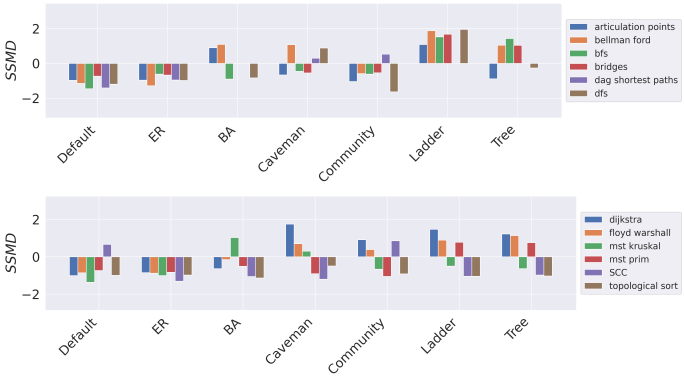
H SSMD for other architectures



(a) Gated MPNN



(b) Gated PGN



(c) Default (Gated triplet-MPNN) with neural priority queue

Figure 8: Adding richer data often does not produce significantly better performance.