Towards Scale-Invariant Graph-related Problem Solving by Iterative Homogeneous Graph Neural Networks

Hao Tang [1] **Zhiao Huang** [2] **Jiayuan Gu** [2] **Bao-Liang Lu** [1] **Hao Su** [2]

Abstract

Current graph neural networks (GNNs) lack generalizability with respect to scales (graph sizes, graph diameters, edge weights, etc..) when solving many graph analysis problems. Taking the perspective of synthesizing graph theory programs, we propose several extensions to address the issue. First, inspired by the dependency of iteration number of common graph theory algorithms on graph sizes, we learn to terminate the message passing process in GNNs adaptively according to the computation progress. Second, inspired by the fact that many graph theory algorithms are homogeneous with respect to graph weights, we introduce homogeneous transformation layers that are universal homogeneous function approximators, to convert ordinary GNNs to be homogeneous. Experimentally, we show that our GNN can be trained from small-scale graphs but generalize well to large-scale graphs for a number of basic graph theory problems. It also shows generalizability for applications of multi-body physical simulation and image-based navigation problems.

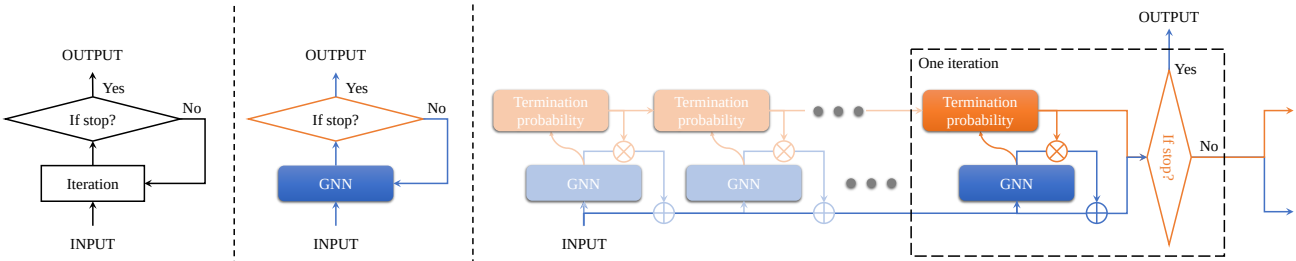
1. Introduction

Graph, as a powerful data representation, arises in many real-world applications (Schlichtkrull et al., 2018; Shang et al., 2019; Fan et al., 2019; Battaglia et al., 2016; Sanchez-Gonzalez et al., 2018; Liu et al., 2018). On the other hand, the flexibility of graphs, including the different representations of isomorphic graphs, the unlimited degree distributions (Muchnik et al., 2013; Seshadri et al., 2008), and the *boundless graph scales* (Ying et al., 2018; Zang et al., 2018), also presents many challenges to their analysis. Recently, Graph Neural Networks (GNNs) have attracted broad attention in solving graph analysis problems. They are permutation-invariant/equivariant by design and have shown superior performance on various graph-based applications (Wu et al., 2020; Zhang et al., 2020; Zhou et al., 2018; Gilmer et al., 2017; Battaglia et al., 2018).

However, investigation into *the generalizability of GNNs with respect to the graph scale* is still limited. Specifically, we are interested in GNNs that can learn from small graphs and perform well on new graphs of arbitrary scales. Existing GNNs (Wu et al., 2020; Zhang et al., 2020; Zhou et al., 2018; Battaglia et al., 2018) are either ineffective or inefficient under this setting. In fact, even ignoring the optimization process of network training, the representation power of existing GNNs is yet too limited to achieve graph scale generalizability. There are at least two issues: 1) By using a pre-defined layer number (Li & Zemel, 2014; Zheng et al., 2015; Tamar et al., 2016), these GNNs are not able to approximate graph algorithms whose complexity depends on graph size (most graph algorithms in textbooks are of this kind). The reason is easy to see: For most GNNs, each node only uses information of the 1-hop neighborhoods to update features by message passing, and it is impossible for k-layer GNNs to send messages between nodes whose distance is larger than k. More formally, Loukas (Loukas, 2020) proves that GNNs, which fall within the message passing framework, lose a significant portion of their power for solving many graph problems when their width and depth are restricted; and 2) a not-so-obvious observation is that, the range of numbers to be encoded by the internal representation may deviate greatly for graphs of different scales. For example, if we train a GNN to solve the shortest path problem on small graphs of diameter k with weight in the range of $[0, 1]$, the internal representation could only need to build the encoding for the path length within $[0, k]$; but if we test this GNN on a large graph of diameter $K \gg k$ with the same weight range, then it has to use and transform the encoding for $[0, K]$. The performance of classical neural network modules (e.g. the multilayer perceptron in GNNs) are usually highly degraded on those out-of-range inputs.

To address the pre-defined layer number issue, we take a program synthesis perspective, to design GNNs that have stronger representation power by mimicking the control flow of classical graph algorithms. Typical graph algorithm, such as the Dijkstra's algorithm for shortest path computation,

[1]Department of Computer Science and Engineering, Shanghai Jiao Tong University, Shanghai, China [2]Department of Computer Science and Engineering, University of California, San Diego, USA. Correspondence to: Hao Tang <tanghaosjtu@gmail.com>, Hao Su <haosu@eng.ucsd.edu>.



(a) iterative algorithms (b) general illustration of Iterative GNN (c) concrete unfolded illustration of Iterative GNN at one iteration

Figure 1. (a) The illustration of general iterative algorithms. The iteration body is repeated until the stopping criterion is satisfied. (b) Illustration of IterGNN as a combination of GNNs and iterative module. (c) A detailed illustration of Iterative GNN. It unfolds the computational flow of IterGNN. Other than the normal data flow (marked as blue), there is another control flow (marked as orange) that serves both as an adaptive stopping criterion and as a data flow controller.

are iterative. They often consist of two sub-modules: an iteration body to solve the sub-problem (e.g., update the distance for the neighborhood of a node as in Dijkstra), and a termination condition to control the loop out of the iteration body. By adjusting the iteration numbers, an iterative algorithm can handle arbitrary large-scale problems. We, therefore, introduce our novel Iterative GNN (IterGNN) that equips ordinary GNN with an adaptive and differentiable stopping criterion to let GNN iterate by itself, as shown in Figure 1. Our stopping condition is adaptive to the inputs, supports arbitrarily large iteration numbers, and interestingly, is able to be trained in an end-to-end fashion *without any direct supervision*.

We also give a partial solution to address the issue of out-of-range number encoding, if the underlying graph algorithm is in a specific hypothesis class. More concretely, the solutions to many graph problems, such as shortest path and TSP, are homogeneous with respect to the input graph weights, i.e., the solution scales linearly with the magnitudes of the input weights. To build GNNs with representation power to approximate the solution to such graph problems, we further introduce the homogeneous inductive-bias. By assuming the message processing functions are homogeneous, the knowledge that neural networks learn at one scale can be generalized to different scales. We build HomoMLP and HomoGNN as powerful approximates of homogeneous functions over vectors and graphs, respectively.

2. Related Works.

We briefly describe the most related ones here due to space limitation and include a complete review in Appendix E. Recently, (Veličković et al., 2020) and (Xu et al., 2020) achieved positive performance on solving the shortest path problem using GNNs. However, (Veličković et al., 2020) requires per-layer supervisions to train the models and methods in (Xu et al., 2020) are short of the generalizability w.r.t. graph scales due to their bounded number of message passing steps. The final formulation of our iterative module is generally similar to the previous adaptive computation time algorithm (ACT) (Graves, 2016) for RNNs or spatially ACT (Figurnov et al., 2017; Eyzaguirre & Soto, 2020) for CNNs, however, with distinct motivations and formulation details. The numbers of iterations for ACT are usually small by design (e.g.the formulation of regularizations and halting distributions). The recent flow-based methods (Poli et al., 2019) , although potentially providing adaptive layer numbers, are not a straightforward solution to approximate iterative algorithms with no explicit iteration controller.

3. Backgrounds

Graphs and graph scales. Each graph $G := (V, E)$ consists of a set of nodes V and a set of edges (pairs of nodes) E. To notate graphs with attributes, we use $\vec{x}_v$ for node attributes of node $v \in V$ and use $\vec{x}_e$ for edge attributes of edge $e \in E$. We consider three graph properties to quantify the graph scales, which are the number of nodes $N := |V|$, which is also called the graph size, and the graph diameter $\delta_G := \max_{u,v \in V} d(u, v)$. Here, $d(u, v)$ denotes the length of the shortest path from node u to node v, which is also called the distance between node u and node v for undirected graphs.

4. Method

We propose Iterative GNN (IterGNN) and Homogeneous GNN (HomoGNN) to improve the generalizability of GNNs with respect to graph scales. IterGNN is first introduced, in Section 4.1, to enable adaptive and unbounded iterations of GNN layers so that the model can generalize to graphs of arbitrary scale. We further introduce HomoGNN, in Section 4.2, to partially solve the problem of out-of-range number encoding, for graph-related problems. We describe

PathGNN layers that improves the generalizability of GNNs for distance-related problems by improving the algorithm alignments (Xu et al., 2020) to the Bellman-Ford algorithm in Appendix B.2.

4.1. Iterative module

Algorithm 1 Iterative module. g is the stopping criterion and f is the iteration body

input: initial feature x; stopping threshold ϵ
$k \leftarrow 1$
$h^0 \leftarrow x$
while $\prod_{i=1}^{k-1}(1-c^i) > \epsilon$ **do**
  $h^k \leftarrow f(h^{k-1})$
  $c^k \leftarrow g(h^k)$
  $k \leftarrow k+1$
end while
return $h = \sum_{k=1}^{\infty} c^k \prod_{i=1}^{k-1}(1-c^i)h^k$

The core of IterGNN is a differentiable iterative module. It executes the same GNN layer repeatedly until a learned stopping criterion is met. We present the pseudo-codes in Algorithm 1. This module includes a stopping criterion function g and an iteration body f. At step k, the iteration body f firstly calculates the new hidden state h^k from previous hidden state h^{k-1}; the stopping criterion function g then calculates a confidence score, which we denote as $c^k \in [0,1]$. The score c^k describes the probability of the iteration to terminate at the current time step. We can then determine the iteration numbers using a random process. In detail, at time step i, we sample from a binomial distribution with probability c^i to decide if the iteration terminates and returns the current hidden states as the output. The probability for the iteration to terminate at time step k is then $p^k = \prod_{i=1}^{k-1}(1-c^i)c^k$, which is also the probability of the network to output h^k in the random process. We take the expectation $E[h] = \sum_{i=1}^{\infty} p^k h^k$ as the output of the iterative module. The gradient to the output h can thus optimize the hidden state h^k and confidence score c^k directly. The iteration stops when the probability $\prod_{i=1}^{k-1}(1-c^i)$, which we also call the left confidence, is smaller than a pre-defined threshold ϵ.

By setting f and g as GNNs, we obtain our novel IterGNN, as shown in Figure 1. The features are associated with nodes in graph as $\{\vec{h}_v^{(k)} : v \in V\}$. GNN layers are adopted as the body function f to update the node features iteratively $\{\vec{h}_v^{(k+1)} : v \in V\} = \text{GNN}(G, \{\vec{h}_v^{(k)} : v \in V\}, \{\vec{h}_e : e \in E\})$. We build the termination probability module as g by integrating a readout function and a MLP. The readout function (e.g. max/mean pooling) summarizes all node features $\{\vec{h}_v^{(k+1)} : v \in V\}$ into a fixed-dimensional vector $\vec{h}^{(k+1)}$. The MLP predicts the confidence score as $c^k = \text{sigmoid}(\text{MLP}(\vec{h}^{(k+1)}))$. The sigmoid function is utilized to ensure the output of g is between 0 and 1. With the help of our iterative module, IterGNN can adaptively adjust the number of iterations. Moreover, it can be trained without any supervision of the stopping condition.

4.2. Homogeneous prior

The homogeneous prior is introduced to improve the generalizability of GNNs for out-of-range number encoding. We first define the positive homogeneous property of a function:

Definition 1. *A function f over vectors is positive homogeneous iff $f(\lambda\vec{x}) = \lambda f(\vec{x})$ for all $\lambda > 0$.*

A function f over graphs is positive homogeneous iff for any graph $G = (V, E)$ with node attributes $\vec{x}_v$ and edge attributes $\vec{x}_e$, $f(G, \{\lambda\vec{x}_v : v \in V\}, \{\lambda\vec{x}_e : e \in E\}) = \lambda f(G, \{\vec{x}_v : v \in V\}, \{\vec{x}_e : e \in E\})$

The solutions to most graph-related problems are positive homogeneous, such as the length of shortest path, the maximum flow, graph radius, and the optimal distance in the travelling salesman problem.

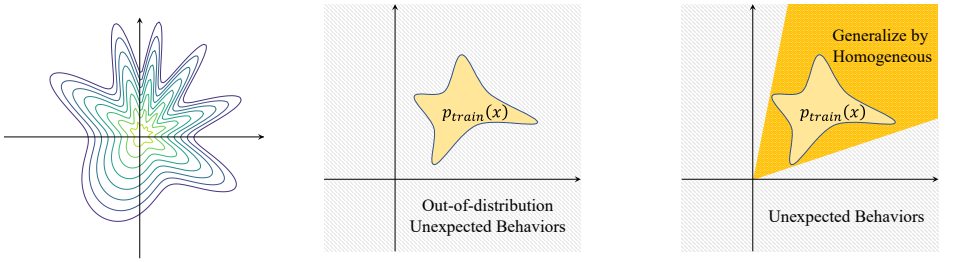
The homogeneous prior tackles the problem of different magnitudes of features for generalization. As illustrated in Figure 2, by assuming functions as positive homogeneous, models can generalize knowledge to the scaled features/attributes of different magnitudes. For example, let's assume two datasets D and D_λ that are only different on magnitudes, which means $D_\lambda := \{\lambda x : x \in D\}$ and $\lambda > 0$. If the target function f and the function $F_{\mathcal{A}}$ represented by neural networks $\mathcal{A}$ are both homogeneous, the prediction error on dataset D_λ then scales linearly with respect to the scaling factor λ:

$$\sum_{x \in D_\lambda} ||f(x) - F_{\mathcal{A}}(x)|| = \lambda \sum_{x' \in D} ||f(x') - F_{\mathcal{A}}(x')||. \quad (1)$$

We design the family of GNNs that are homogeneous, named HomoGNN, as follows: simply remove all the bias terms in the multi-layer perceptron (MLP) used by ordinary GNNs, so that all affine transformations degenerate to linear transformations. Additionally, only activation functions that are homogeneous are allowed to be used. Note that ReLU is a homogeneous activation function. The original MLP used in ordinary GNNs become HomoMLP in HomoGNNs afterwards.

5. Experiments

We evaluate the generalizability of models with respect to graph scales on three graph theory problems, i.e. shortest path, component counting, and Traveling Salesman Problem (TSP). We build a benchmark by combining multiple



(a) $f(\lambda x) = \lambda f(x),\ \forall \lambda > 0, x \in R^2$ (b) Behaviors of MLP (c) Behaviors of Homogeneous MLP

Figure 2. (a) An example of homogeneous functions. (b-c) Illustration of the improved generalizability by applying the homogeneous prior. Knowledge learned from the training samples not only can be generalized to samples of the same data distribution as ordinary neural networks, as shown in (b), but also can be generalized to samples of the scaled data distributions, as shown in (c).

Table 1. Generalization performance on graph algorithm learning . Models are trained on graphs of sizes within $[4, 34)$ and are tested on graph of larger sizes such as 100 and 500. The metric for weighted shortest path and TSP is the relative loss. The metric for component counting is the accuracy. The metric for the unweighted shortest path problem is the success rate of identifying the shortest paths.

	Shortest Path - weighted				Component Cnt.		TSP	Shortest Path - unweighted				
Models	ER	PL	KNN	Lob	ER	Lob	2D	20	100	500	1000	5000
GCN	0.1937	0.202	0.44	0.44	0.0%	0.0%	0.52	66.6	25.7	5.5	2.4	0.4
GAT	0.1731	0.127	0.26	0.28	0.0%	0.0%	0.18	**100.0**	42.7	10.5	5.3	0.9
Path (ours)	0.0014	0.084	0.16	0.29	82.3%	36.9%	0.16	**100.0**	62.9	20.1	10.3	1.6
Homo-Path (ours)	**0.0008**	0.015	0.07	0.27	**91.9%**	83.9%	0.14	**100.0**	72.7	58.3	53.7	50.2
Iter-Homo-Path (ours)	**0.0007**	**0.003**	**0.03**	**0.02**	86.6%	**95.4%**	**0.11**	**100.0**	**100.0**	**100.0**	**100.0**	**100.0**

graph generators, including Erdos-Renyi (ER), K-Nearest-Neighborhoods graphs (KNN), planar graphs (PL), and lobster graphs (Lob), so that the generated graphs can have more diverse properties. The generation processes and the properties of datasets are listed in the Appendix C.

Models and baselines. We stack 30 GCN (Kipf & Welling, 2017)/GAT (Veličković et al., 2018) layers to build the baseline models. GIN (Xu et al., 2019) is not enlisted since 30-layer GINs do not converge in most of our preliminary experiments. Our "Path" model stacks 30 PathGNN layers. Our "Homo-Path" model replaces GNNs and MLPs in the "Path" model with HomoGNNs and HomoMLPs. The "Iter-Homo-Path" model adopts the iterative module to control the iteration number of the GNN layer in the "Homo-Path" model.

Training Details. We utilize the default hyper-parameters to train models. We generate 10000 samples for training, 1000 samples for validation, and 1000 samples for testing. The only two tunable hyper-parameter in our experiment is the epoch number (10 choices) and the formulation of PathGNN layers (3 choices). Validation datasets are used to tune them. More details are presented in Appendix C.5.

Results and analysis. We present the generalization performance for all three graph theory problems in Table 1. Models are trained on graphs of sizes within $[4, 34)$ and are evaluated on graphs of larger sizes such as 100 (for shortest path and TSP) and 500 (for component counting so that the diameters of components are large enough). The relative loss metric is defined as $|y - \hat{y}|/|y|$, given a label y and a prediction $\hat{y}$. The results demonstrate that each of our proposals improves the generalizability on almost all problems. Exceptions happen on graphs generated by ER. It is because the diameters of those graphs are 2 with high probability even though the graph sizes are large. Our final model, Iter-Homo-Path, which integrates all proposals, performs much better than the baselines such as GCN and GAT.

We then explore models' generalizability on much larger graphs on the shortest path problem using Lob to generate graphs with larger diameters. As shown in Table 1, our model achieves 100% success rate of identifying the shortest paths on graphs with as large as 5000 nodes even though they are trained on graphs of sizes within $[4, 34)$. As claimed, the iterative module is necessary for generalizing to graphs of much larger sizes and diameters due to the message passing nature of GNNs.

6. Conclusion

We propose IterGNN and HomoGNN to improve the generalizability of GNNs w.r.t. graph scales. Experiments show that our proposals do improve the generalizability for solving multiple graph-related problems and tasks.

Acknowledgements

B. L. Lu was supported in part by the National Key Research and Development Program of China (2017YFB1002501), the National Natural Science Foundation of China (61673266 and 61976135), SJTU Trans-med Awards Research (WF540162605), the Fundamental Research Funds for the Central Universities, and the 111 Project. We specially thank Zhizuo Zhang, Zhiwei Jia, and Chutong Yang for the extremely useful discussions, and Wei-Long Zheng, Bingyu Shen and Yuming Zhao for reviewing the paper prior to submission.

References

Applegate, D. L., Bixby, R. E., Chvatal, V., and Cook, W. J. http://www.math.uwaterloo.ca/tsp/concorde.html.

Araujo, F., Ribeiro, B., and Rodrigues, L. A neural network for shortest path computation. *IEEE Transactions on Neural Networks*, 12(5):1067–1073, 2001.

Barber, C. B., Dobkin, D. P., and Huhdanpaa, H. The quickhull algorithm for convex hulls. *ACM Transactions on Mathematical Software (TOMS)*, 22(4):469–483, 1996.

Battaglia, P., Pascanu, R., Lai, M., Rezende, D. J., et al. Interaction networks for learning about objects, relations and physics. In *Advances in Neural Information Processing Systems*, pp. 4502–4510, 2016.

Battaglia, P. W., Hamrick, J. B., Bapst, V., Sanchez-Gonzalez, A., Zambaldi, V., Malinowski, M., Tacchetti, A., Raposo, D., Santoro, A., Faulkner, R., et al. Relational inductive biases, deep learning, and graph networks. *arXiv preprint arXiv:1806.01261*, 2018.

Bellemare, M. G., Naddaf, Y., Veness, J., and Bowling, M. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research*, 47:253–279, jun 2013.

Blei, D. M., Kucukelbir, A., and McAuliffe, J. D. Variational inference: A review for statisticians. *Journal of the American statistical Association*, 112(518):859–877, 2017.

Cangea, C., Veličković, P., Jovanović, N., Kipf, T., and Liò, P. Towards sparse hierarchical graph classifiers. *the 32nd Annual Conference on Neural Information Processing Systems (NeurIPS)*, 2018.

Cybenko, G. Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems*, 2(4):303–314, 1989.

Dai, H., Kozareva, Z., Dai, B., Smola, A., and Song, L. Learning steady-states of iterative algorithms over graphs. In *International Conference on Machine Learning*, pp. 1114–1122, 2018.

Dijkstra, E. W. A note on two problems in connexion with graphs. *Numerische mathematik*, 1(1):269–271, 1959.

Erdos, P. and Renyi, A. On random graphs. *Publ. Math. Debrecen*, 6:290–297, 1959.

Eyzaguirre, C. and Soto, A. Differentiable adaptive computation time for visual reasoning. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2020.

Fan, W., Ma, Y., Li, Q., He, Y., Zhao, E., Tang, J., and Yin, D. Graph neural networks for social recommendation. In *The World Wide Web Conference*, pp. 417–426. ACM, 2019.

Figurnov, M., Collins, M. D., Zhu, Y., Zhang, L., Huang, J., Vetrov, D., and Salakhutdinov, R. Spatially adaptive computation time for residual networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 1039–1048, 2017.

Gao, H. and Ji, S. Graph u-nets. *International Conference on Machine Learning (ICML)*, 2019.

Gilmer, J., Schoenholz, S. S., Riley, P. F., Vinyals, O., and Dahl, G. E. Neural message passing for quantum chemistry. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pp. 1263–1272. JMLR. org, 2017.

Goyal, P. and Ferrara, E. Graph embedding techniques, applications, and performance: A survey. *Knowledge-Based Systems*, 151:78–94, 2018.

Graves, A. Adaptive computation time for recurrent neural networks. *arXiv preprint arXiv:1603.08983*, 2016.

Graves, A., Wayne, G., and Danihelka, I. Neural turing machines. *arXiv preprint arXiv:1410.5401*, 2014.

Graves, A., Wayne, G., Reynolds, M., Harley, T., Danihelka, I., Grabska-Barwińska, A., Colmenarejo, S. G., Grefenstette, E., Ramalho, T., Agapiou, J., et al. Hybrid computing using a neural network with dynamic external memory. *Nature*, 538(7626):471, 2016.

Huang, Z., Liu, F., and Su, H. Mapping state space using landmarks for universal goal reaching. In *Advances in Neural Information Processing Systems*, pp. 1940–1950, 2019.

Kipf, T. N. and Welling, M. Semi-supervised classification with graph convolutional networks. In *International Conference on Learning Representations (ICLR)*, 2017.

Lee, J., Lee, I., and Kang, J. Self-attention graph pooling. In *International Conference on Machine Learning (ICML)*, 2019.

Li, Y. and Zemel, R. S. Mean field networks. *ICML workshop on Learning Tractable Probabilistic Models*, 2014. URL https://academic.microsoft.com/paper/2271601362.

Li, Y., Tarlow, D., Brockschmidt, M., and Zemel, R. Gated graph sequence neural networks. In *International Conference on Learning Representations*, 2016.

Liu, X., Luo, Z., and Huang, H. Jointly multiple events extraction via attention-based graph information aggregation. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, pp. 1247–1256, 2018. URL https://aclanthology.info/papers/D18-1156/d18-1156.

Loukas, A. What graph neural networks cannot learn: depth vs width. In *International Conference on Learning Representations*, 2020. URL https://openreview.net/forum?id=B1l2bp4YwS.

Muchnik, L., Pei, S., Parra, L. C., Reis, S. D., Andrade Jr, J. S., Havlin, S., and Makse, H. A. Origins of power-law degree distribution in the heterogeneity of human activity in social networks. *Scientific reports*, 3:1783, 2013.

Neyshabur, B., Bhojanapalli, S., Mcallester, D., and Srebro, N. Exploring generalization in deep learning. In Guyon, I., Luxburg, U. V., Bengio, S., Wallach, H., Fergus, R., Vishwanathan, S., and Garnett, R. (eds.), *Advances in Neural Information Processing Systems 30*, pp. 5947–5956. Curran Associates, Inc., 2017.

Page, L., Brin, S., Motwani, R., and Winograd, T. The pagerank citation ranking: Bringing order to the web. Technical report, Stanford InfoLab, 1999.

Poli, M., Massaroli, S., Park, J., Yamashita, A., Asama, H., and Park, J. Graph neural ordinary differential equations. *arXiv preprint arXiv:1911.07532*, 2019.

Qi, C. R., Su, H., Mo, K., and Guibas, L. J. Pointnet: Deep learning on point sets for 3d classification and segmentation. In *Proceedings of the IEEE conference on Computer Vision and Pattern Recognition*, pp. 652–660, 2017.

Sanchez-Gonzalez, A., Heess, N., Springenberg, J. T., Merel, J., Riedmiller, M., Hadsell, R., and Battaglia, P. Graph networks as learnable physics engines for inference and control. In Dy, J. and Krause, A. (eds.), *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pp. 4470–4479, Stockholmsmässan, Stockholm Sweden, 10–15 Jul 2018. PMLR. URL http://proceedings.mlr.press/v80/sanchez-gonzalez18a.html.

Schlichtkrull, M., Kipf, T. N., Bloem, P., Van Den Berg, R., Titov, I., and Welling, M. Modeling relational data with graph convolutional networks. In *European Semantic Web Conference*, pp. 593–607. Springer, 2018.

Seshadri, M., Machiraju, S., Sridharan, A., Bolot, J., Faloutsos, C., and Leskove, J. Mobile call graphs: beyond power-law and lognormal distributions. In *Proceedings of the 14th ACM SIGKDD international conference on Knowledge Discovery and Data Mining*, pp. 596–604. ACM, 2008.

Shang, C., Tang, Y., Huang, J., Bi, J., He, X., and Zhou, B. End-to-end structure-aware convolutional networks for knowledge base completion. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pp. 3060–3067, 2019.

Tamar, A., Wu, Y., Thomas, G., Levine, S., and Abbeel, P. Value iteration networks. In *Advances in Neural Information Processing Systems*, pp. 2154–2162, 2016.

Van Hasselt, H., Guez, A., and Silver, D. Deep reinforcement learning with double q-learning. In *Thirtieth AAAI conference on artificial intelligence*, 2016.

Veličković, P., Cucurull, G., Casanova, A., Romero, A., Liò, P., and Bengio, Y. Graph attention networks. In *International Conference on Learning Representations*, 2018. URL https://openreview.net/forum?id=rJXMpikCZ.

Veličković, P., Ying, R., Padovano, M., Hadsell, R., and Blundell, C. Neural execution of graph algorithms. In *International Conference on Learning Representations*, 2020. URL https://openreview.net/forum?id=SkgKO0EtvS.

Wu, Z., Pan, S., Chen, F., Long, G., Zhang, C., and Yu, P. S. A comprehensive survey on graph neural networks. *IEEE Transactions on Neural Networks and Learning Systems*, pp. 1–21, 2020.

Xu, K., Li, C., Tian, Y., Sonobe, T., Kawarabayashi, K.-i., and Jegelka, S. Representation learning on graphs with jumping knowledge networks. In *International Conference on Machine Learning (ICML)*, 2018.

Xu, K., Hu, W., Leskovec, J., and Jegelka, S. How powerful are graph neural networks? In *International Conference on Learning Representations*, 2019. URL https://openreview.net/forum?id=ryGs6iA5Km.

Xu, K., Li, J., Zhang, M., Du, S. S., ichi Kawarabayashi, K., and Jegelka, S. What can neural networks reason about? In *International Conference on Learning Representations*, 2020. URL https://openreview.net/forum?id=rJxbJeHFPS.

Ying, R., He, R., Chen, K., Eksombatchai, P., Hamilton, W. L., and Leskovec, J. Graph convolutional neural networks for web-scale recommender systems. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pp. 974–983. ACM, 2018.

Zang, C., Cui, P., Faloutsos, C., and Zhu, W. On power law growth of social networks. *IEEE Transactions on Knowledge and Data Engineering*, 30(9):1727–1740, Sep. 2018. ISSN 1041-4347. doi: 10.1109/TKDE.2018.2801844.

Zhang, M., Cui, Z., Neumann, M., and Chen, Y. An end-to-end deep learning architecture for graph classification. In *AAAI*, pp. 4438–4445, 2018.

Zhang, Z., Cui, P., and Zhu, W. Deep learning on graphs: A survey. *IEEE Transactions on Knowledge and Data Engineering*, pp. 1–1, 2020.

Zheng, S., Jayasumana, S., Romera-Paredes, B., Vineet, V., Su, Z., Du, D., Huang, C., and Torr, P. H. Conditional random fields as recurrent neural networks. In *Proceedings of the IEEE international conference on computer vision*, pp. 1529–1537, 2015.

Zhou, J., Cui, G., Zhang, Z., Yang, C., Liu, Z., and Sun, M. Graph neural networks: A review of methods and applications. *arXiv preprint arXiv:1812.08434*, 2018.

Appendix

In the appendix, we first present the theoretical analysis of our proposals, i.e. the iterative module and the homogeneous prior. We seek to answer to the following questions: **(1)** How powerful is our iterative module on approximating the iterative algorithms? **(2)** Are low generalization errors achievable when using homogeneous neural networks to approximate the homogeneous functions? What is the generation error bound? **(3)** Are low training errors achievable when using HomoMLP to approximate the homogeneous functions? Is HomoMLP a universal approximator of positive homogeneous functions?

To answer the question **(1)**, we present the theoretical analysis of the representation power of our iterative module in Section A.1. To answer the question **(2)**, we prove the generalization error bounds of homogeneous neural networks on approximating homogeneous functions with independent scaling assumptions in Section A.2.1. A concrete bound based on the PAC-Bayesian framework is also presented in Lemma A.2.1. To answer the question **(3)**, we present and prove the universal approximation theorem on approximating the homogeneous functions for HomoMLP in Section A.2.3.

Furthermore, we show more experimental results to answer questions: **(4)** How does each of our proposals improve the generalizability w.r.t. graph scales? How many improvements do our proposals achieve compared with the baselines? **(5)** Will our iterative module adaptively change the iteration numbers and consequently learn an interpretable stopping criterion in practice? **(6)** Can our proposals improve the performance of general graph-based reasoning tasks such as those in physical simulation, image-based navigation, and reinforcement learning? **(7)** Is the iterative module harmful to the standard generalizability in practice? What is its performance on graph-classification benchmarks?

To answer question **(4)**, we perform the ablation studies in Section D.1.1. To answer question **(5)**, we show that our iterative module actually learns an optimal and interpretable stopping criterion in Section D.1.2. To answer question **(6)**, we evaluate our models on three general reasoning tasks in Section C.2 and Section D.2. To answer question **(7)**, we show that our IterGIN model, which wraps each layer of the state-of-art GIN (Xu et al., 2019) model with our iterative module, achieves competitive performance to GIN, in Section D.3.

In general, we provide the theoretical analysis of our proposals in Section A. We describe the detailed formulations of our proposals in Section B. The omitted experimental setups are all listed in Section C and the omitted experimental results are presented in Section D. At last, we also state more background knowledges of graph neural networks (GNNs) in Section F.

A. Theoretical analysis

We present the theoretical analysis of our proposals in this section. In detail, we analyze the representation power of our iterative module in Section A.1. We prove the generalization error bounds of homogeneous neural networks in Section A.2.1. We show that HomoMLP and HomoGNN can only represent homogeneous functions in Section A.2.2. We also prove the universal approximation theorem of HomoMLP on approximating homogeneous functions in Section A.2.3.

A.1. Representation powers of iterative module

We first state the intuition that our iterative module as described in the main body can approximate any iterative algorithms as defined in Algorithm 2, as long as the $body$ and $condition$ functions are available or can be perfectly reproduced by neural networks.

Algorithm 2 Iterative algorithm

input initial feature x
$k \leftarrow 1$
$h^0 \leftarrow x$
while not $condition(h^k)$ **do**
  $h^k \leftarrow body(h^{k-1})$
  $k \leftarrow k + 1$
end while
return $h = h^k$

More formally, we build an ideal class of models, named as Iter-Oracle, by combining our iterative module with the oracles $\mathcal{F}_\theta$ that can perfectly reproduce the $body$ function and the $condition$ function, which means there exist θ' and θ'' such that

for all x, $\mathcal{F}_{\theta'}(x) = body(x)$ and $\mathcal{F}_{\theta''}(x) = condition(x)$. We can then show the representation power of our iterative module using the following theorem:

Theorem A.1.1. *For any iterative algorithm, iter-alg, defined as in Algorithm 2, for any initial feature x, and for any $\epsilon > 0$, there exist an Iter-Oracle model $\mathcal{A}$, which represents the function $\mathcal{F}_{\mathcal{A}}$, satisfying*

$$||\textit{iter-alg}(x) - \mathcal{F}_{\mathcal{A}}(x)|| < \epsilon. \tag{2}$$

We prove it by construction. We build the function f in our iterative module using $\mathcal{F}_{\theta'}$ such that $\mathcal{F}_{\theta'}(x) = body(x)$ for all x. The function g in our iterative module is built as sigmoid$(\alpha(\mathcal{F}_{\theta''}(x) - 0.5))$, where $\mathcal{F}_{\theta''}(x) = condition(x)$ for all x and we utilize similar rules to the python language for type conversion, which means $condition(x)$ outputs 1 if the condition is satisfied and outputs 0 otherwise. By setting $\alpha \to +\infty$, we have

$$c^k \to \begin{cases} 1 & \text{if } condition(h^k) \\ 0 & \text{if not } condition(h^k). \end{cases}$$

Therefore, $\sum_{j=1}^{\infty} c^j h^j \prod_{i=1}^{j-1}(1-c^i) \to h^k$, where k is the time step that $condition(h^k)$ is firstly satisfied. More formally, assume Λ bounds the norm of features h^j for the specific iterative algorithm $iter\text{-}alg$, for the specific initial feature x, and for any j, we can set α as

$$\alpha > 2\ln\left(\frac{(k+1)\Lambda}{\epsilon} - 1\right) \text{ and } \alpha > -2\ln\left(\left(1 + \frac{\epsilon}{(k+1)\Lambda}\right)^{-\frac{1}{k}} - 1\right), \tag{3}$$

so that Eq. 2 is satisfied, since for $j < k$,

$$\left\| c^j h^j \prod_{i=1}^{j-1}(1-c^i) - 0 \right\| < \left\| c^j h^j \right\| < \frac{1}{1+e^{\frac{\alpha}{2}}}\Lambda = \frac{\epsilon}{k+1}, \tag{4}$$

for $j = k$,

$$\left\| c^k h^k \prod_{i=1}^{k-1}(1-c^i) - h^k \right\| < \left(\left(\frac{1}{1+e^{-\frac{\alpha}{2}}}\right)^k - 1\right)\Lambda = \frac{\epsilon}{k+1}, \tag{5}$$

and for $j > k$,

$$\left\| \sum_{j=k+1}^{\infty} c^j h^j \prod_{i=1}^{j-1}(1-c^i) - 0 \right\| < (1-c^k)\Lambda < \frac{1}{1+e^{\frac{\alpha}{2}}}\Lambda = \frac{\epsilon}{k+1}. \tag{6}$$

Together, we have

$$\begin{aligned} \left\| \sum_{j=1}^{\infty} c^j h^j \prod_{i=1}^{j-1}(1-c^i) - h^k \right\| \quad &\leq \quad \sum_{j=1}^{k-1} \left\| c^j h^j \prod_{i=1}^{j-1}(1-c^i) - 0 \right\| + & (7) \\ &\quad \left\| c^k h^k \prod_{i=1}^{k-1}(1-c^i) - h^k \right\| + & (8) \\ &\quad \left\| \sum_{j=k+1}^{\infty} c^j h^j \prod_{i=1}^{j-1}(1-c^i) - 0 \right\| & (9) \\ &< \quad (k+1)\frac{\epsilon}{k+1} = \epsilon \quad \square \end{aligned}$$

We then derive a more practical proposition of IterGNN, based on Theorem A.1.1, stating the intuition that IterGNN can achieve adaptive and unbounded iteration numbers:

Proposition A.1.1. *Under the assumptions that IterGNN can calculate graph sizes N with no error and the multilayer perceptron used by IterGNN is a universal approximator of continuous functions on compact subsets of $\mathbb{R}^n$ ($n \geq 1$) (i.e. the universal approximation theorem), there exist IterGNNs whose iteration numbers are constant, linear, polynomial or exponential functions of the graph sizes.*

The proofs are simple. Let g' be the function that maps the graph sizes N to iteration numbers k. we can then build GNNs as f to calculate the graph sizes N and the number of current time step j, and build g as sigmoid$(\alpha(0.5 - |g'(N) - j|))$. The α can be set similarly to the previous proof except for one scalar that compensates the approximation errors of neural networks. More formally, assume ϵ' bounds the error of predicting $g'(N)$ and j using neural networks $\mathcal{A}$, which means for any input, there exists neural networks that represent functions $\mathcal{F}_g$ and $\mathcal{F}_j$ satisfying $|g'(N) - \mathcal{F}_g| < \epsilon'$ and $|j - \mathcal{F}_j| < \epsilon'$. We can then set α as

$$\alpha > \frac{2}{1-4\epsilon'} \ln\left(\frac{(k+1)\Lambda}{\epsilon} - 1\right) \text{ and } \alpha > -\frac{2}{1-4\epsilon'} \ln\left(\left(1 + \frac{\epsilon}{(k+1)\Lambda}\right)^{-\frac{1}{k}} - 1\right), \tag{10}$$

so that Proposition A.1.1 is satisfied. Note that it is easy to build GNNs to exactly calculate the graph sizes N and the number of current time step j. Given the universal approximation theorem of MLP, the stopping condition function g can also be easily approximated by MLPs. Our iterative module is thus able to achieve adaptive and unbounded iteration numbers using neural networks.

A.2. Homogeneous prior

We formalize the generalization error bounds of homogeneous neural networks on approximating homogeneous functions under proper conditions, by extending the example in the main body to more general cases, in Section A.2.1. To make sure functions represented by neural networks are homogeneous, we also prove that HomoGNN and HomoMLP can only represent homogeneous functions, in Section A.2.2. To show that low training errors are achievable, we further analyze the representation powers of HomoMLP and demonstrate that it is a universal approximator of homogeneous functions under some assumptions, based on the universal approximation theorem of MLPs (Cybenko, 1989), in Section A.2.3.

A.2.1. Generalization error bounds of homogeneous neural networks

Extending the example in the main body to more general cases, we present the out-of-distribution generalization error bounds of homogeneous neural networks on approximating homogeneous functions under the assumption of independent scaling of magnitudes during inference:

Let training samples $D_m = \{x_1, x_2, \cdots x_m\}$ be independently sampled from the distribution $\mathcal{D}_x$, then if we scale the training samples with the scaling factor $\lambda \in \mathbb{R}^+$ which is independently sampled from the distribution $\mathcal{D}_\lambda$, we get a "scaled" distribution $\mathcal{D}_x^\lambda$, which has a density function $P_{\mathcal{D}_x^\lambda}(z) := \int_\lambda \int_x \delta(\lambda x = z) P_{\mathcal{D}_\lambda}(\lambda) P_{\mathcal{D}_x}(x) \,\mathrm{d}x \,\mathrm{d}\lambda$. The following theorem bounds the generalization error bounds on $\mathcal{D}_x^\lambda$:

Theorem A.2.1. (Generalization error bounds of homogeneous neural networks with independent scaling assumption). *For any positive homogeneous functions function f and neural network $F_\mathcal{A}$, let β bounds the generalization errors on the training distribution D_x , i.e., $\mathbb{E}_{x\sim\mathcal{D}_x}|f(x) - \mathcal{F}_\mathcal{A}(x)| \leq \frac{1}{m}\sum_{i=1}^m |f(x_i) - F_\mathcal{A}(x_i)| + \beta$, then the generalization errors on the scaled distributions $\mathcal{D}_x^\lambda$ scale linearly with the expectation of scales $\mathbb{E}_{\mathcal{D}_\lambda}[\lambda]$:*

$$\mathbb{E}_{x\sim\mathcal{D}_x^\lambda}|f(x) - F_\mathcal{A}(x)| = \mathbb{E}_{\mathcal{D}_\lambda}[\lambda]\mathbb{E}_{x\sim\mathcal{D}_x}|f(x) - F_\mathcal{A}(x)| \leq \mathbb{E}_{\mathcal{D}_\lambda}[\lambda](\frac{1}{m}\sum_{i=1}^m |f(x_i) - F_\mathcal{A}(x_i)| + \beta) \tag{11}$$

The proof is as simple as re-expressing the formulas:

$$\begin{aligned}
\mathbb{E}_{z\sim\mathcal{D}_x^\lambda}|f(z)-F_{\mathcal{A}}(z)| &= \int_\lambda\int_x P_{\mathcal{D}_\lambda}(\lambda)P_{\mathcal{D}_x}(x)|f(\lambda x)-F_{\mathcal{A}}(\lambda x)|\,\mathrm{d}x\,\mathrm{d}\lambda &(12)\\
&= \int_\lambda P_{\mathcal{D}_\lambda}(\lambda)\lambda\,\mathrm{d}\lambda\int_x P_{\mathcal{D}_x}(x)|f(x)-F_{\mathcal{A}}(x)|\,\mathrm{d}x &(13)\\
&= \mathbb{E}_{\mathcal{D}_\lambda}[\lambda]\mathbb{E}_{x\sim\mathcal{D}_x}|f(x)-F_{\mathcal{A}}(x)| &(14)\\
&\le \mathbb{E}_{\mathcal{D}_\lambda}[\lambda](\frac{1}{m}\sum_{i=1}^m|f(x_i)-F_{\mathcal{A}}(x_i)|+\beta) \quad \square
\end{aligned}$$

Theorem A.2.1 can be considered as a meta-bound and can thus be integrated with any specific generalization error bounds with classical i.i.d. assumptions to create a concrete generation error bounds of homogeneous neural networks on approximating homogeneous functions with independent scaling assumptions. For example, when integrated with a generation error bounds in the PAC-Bayesian framework (Eq.7 in (Neyshabur et al., 2017)), we obtain the following lemma: Let $f_{\boldsymbol{w}}$ be any predictor learned from training data. We consider a distribution $\mathcal{Q}$ over predictors with weights of the form $\boldsymbol{w}+\boldsymbol{v}$, where $\boldsymbol{w}$ is a single predictor learned from the training set, and $\boldsymbol{v}$ is a random variable.

Lemma A.2.1. *Assume all hypothesis h and $f_{\boldsymbol{w}+\boldsymbol{v}}$ for any $\boldsymbol{v}$ are positive homogeneous functions, as defined in Definition 1. Then, given a "prior" distribution P over the hypothesis that is independent of the training data, with probability at least $1-\delta$ over the draw of the training data, the expected error of $f_{\boldsymbol{w}+\boldsymbol{v}}$ on the scaled distribution $\mathcal{D}_x^\lambda$ can be bounded as follows*

$$\mathbb{E}_{\boldsymbol{v}}\left[\mathbb{E}_{x\sim\mathcal{D}_x^\lambda}\left[\left|f(x)-f_{\boldsymbol{w}+\boldsymbol{v}}(x)\right|\right]\right] \le \mathbb{E}_{\mathcal{D}_\lambda}[\lambda]\left(\mathbb{E}_{\boldsymbol{v}}\left[\frac{1}{m}\sum_{i=1}^m\left|f(x_i)-f_{\boldsymbol{w}+\boldsymbol{v}}(x_i)\right|\right]+4\sqrt{\frac{1}{m}\left(KL(\boldsymbol{w}+\boldsymbol{v}||P)+\ln\frac{2m}{\delta}\right)}\right).$$

A.2.2. HomoMLP and HomoGNN are homogeneous functions

We present that HomoGNN and HomoMLP can only represent homogeneous functions:

Proposition A.2.1. *For any input x, we have HomoMLP$(\lambda x)=\lambda$HomoMLP(x) for all $\lambda>0$.*

Proposition A.2.2. *For any graph $G=(V,E)$ with node attributes $\vec{x}_v$ and edge attributes $\vec{x}_e$, we have for all $\lambda>0$,*

$$\textit{HomoGNN}(G,\{\lambda\vec{x}_v:v\in V\},\{\lambda\vec{x}_e:e\in E\})=\lambda\textit{HomoGNN}(G,\{\vec{x}_v:v\in V\},\{\vec{x}_e:e\in E\}). \tag{15}$$

Both propositions are derived from the following lemma:

Lemma A.2.2. *Compositions of homogeneous functions are homogeneous functions.*

We compose functions by taking the outputs of functions as the input of other functions. The inputs of functions are either the outputs of other functions or the initial features x. For example, we can compose functions f,g,h as $h(f(x),g(x),x)$. If f,g,h are all homogeneous functions, we have for all x and all $\lambda>0$,

$$h(f(\lambda x),g(\lambda x),\lambda x)=\lambda h(f(x),g(x),x). \tag{16}$$

More formally, we can prove the lemma by induction. We denote the composition of a set of functions $\{f_1,f_2,\cdots,f_n\}$ as $O(\{f_1,f_2,\cdots,f_n\})$. Note that there are multiple ways to compose n functions. Here, $O(\{f_1,f_2,\cdots,f_n\})$ just denotes one possible way of composition, and we can use $O'(\{f_1,f_2,\cdots,f_n\})$ to denote another. We want to prove that the composition of homogeneous functions is still a homogeneous function, which means for all $n>0$, for all $\lambda>0$, and for all possible ways of compositions O, $O(\{f_1,f_2,\cdots,f_n\})(\lambda x)=\lambda O(\{f_1,f_2,\cdots,f_n\})(x)$.

The base case: $n=1$. The composition of a single function $O(\{f_1\})$ is itself f_1. Therefore, $O(\{f_1\})$ is homogeneous by definition as $O(\{f_1\})(\lambda x)=f_1(\lambda x)=\lambda O(\{f_1\})(x)$.

Assume the composition of the k functions is homogeneous, for any composition of $k+1$ functions $O(\{f_1,f_2,\cdots,f_{k+1}\})$, let f_{k+1} be the last function of the compositions, which means $O(\{f_1,f_2,\cdots,f_{k+1}\})(x):=$

$f_{k+1}(O'(\{f_1, f_2, \cdots, f_k\})(x), O''(\{f_1, f_2, \cdots, f_k\})(x), O'''(\{f_1, f_2, \cdots, f_k\})(x), \cdots)$, it's easy to see that, according to the definition of homogeneous functions,

$$O(\{f_1, f_2, \cdots, f_{k+1}\})(\lambda x) = \lambda O(\{f_1, f_2, \cdots, f_{k+1}\})(x). \quad \square$$

Proposition A.2.1 and Proposition A.2.1 can all be considered as specializations of Lemma A.2.2.

A.2.3. Universal approximation theorem of HomoMLP

We first introduce a class of neural networks that can be proved as a universal approximator of homogeneous functions (Theorem A.2.2) and then show that our HomoMLP is as powerful as this class of neural networks to prove that our HomoMLP is a universal approximator of homogeneous functions (Theorem A.2.3).

We construct a class of neural networks as the universal approximator of positive homogeneous functions, as defined in the main body, as follows:

$$G_{\text{MLP}}(x) = |x| \mathcal{F}_{\text{MLP}}(\frac{x}{|x|}), \tag{17}$$

where $|\cdot|$ denotes the L1 norm, the MLP denotes the classical multilayer perceptrons with positive homogeneous activation functions, and $\mathcal{F}_{\text{MLP}}$ is the function represented by the specific MLP.

Proposition A.2.3. *For any input x, we have $G_{MLP}(\lambda x) = \lambda G_{MLP}(x)$.*

This proposition can be easily proved by re-expressing the formulas:

$$G_{\text{MLP}}(\lambda x) = |\lambda x| \mathcal{F}_{\text{MLP}}(\frac{\lambda x}{|\lambda x|}) = \lambda |x| \mathcal{F}_{\text{MLP}}(\frac{x}{|x|}) = \lambda G_{\text{MLP}}(x) \quad \square$$

We show that it is a universal approximator of homogeneous functions: Let $\mathbb{X}$ be a compact subset of $\mathbb{R}^m$ and $C(\mathbb{X})$ denotes the space of real-valued continuous functions on $\mathbb{X}$.

Theorem A.2.2. (Univeral approximation theorem for G_{MLP}.) *Given any $\epsilon > 0$ and any function $f \in C(\mathbb{X})$, if the function f is positive homogeneous as defined in Definition 1, there exist a finite-layer feed-forward neural networks $\mathcal{A}$ with positive homogeneous activation functions such that for all $x \in \mathbb{X}$, we have $|G_{\mathcal{A}}(x) - f(x)| = \left| |x| \, \mathcal{F}_{\mathcal{A}} \left(\frac{x}{|x|} \right) - f\left(x\right) \right| < \epsilon$.*

The prove is as simple as applying the universal approximation theorem of MLPs (Cybenko, 1989) and applying the definition of the homogeneous functions. In detail, as $\mathbb{X}$ is a compact subset, the magnitudes of inputs x is bounded. We use M to denote the bound, which means $|x| \leq M$ for all $x \in \mathbb{X}$. According to the universal approximation theorem of MLPs (Cybenko, 1989), there exists a finite-layer feed-forward layer $\mathcal{A}$ with ReLU as activation functions such that $\left| \mathcal{F}_{\mathcal{A}} \left(\frac{x}{|x|} \right) - f\left(\frac{x}{|x|} \right) \right| < \frac{\epsilon}{M}$ for all $x \in \mathbb{X}$. Then, according to the definition of homogeneous functions, we have

$$|G_{\mathcal{A}}(x) - f(x)| = \left| |x| \, \mathcal{F}_{\mathcal{A}} \left(\frac{x}{|x|} \right) - |x| \, f\left(\frac{x}{|x|} \right) \right| < |x| \, \frac{\epsilon}{M} < \epsilon \tag{18}$$

Note that ReLU is positive homogeneous. Therefore, we finish the proof of Theorem A.2.2. $\square$

We then prove that HomoMLP is a universal approximator of homogeneous functions: Let $\mathbb{X}$ be a compact subset of $\mathbb{R}^m$ and $C(\mathbb{X})$ denotes the space of real-valued continuous functions on $\mathbb{X}$.

Theorem A.2.3. (Universal approximation theorem for HomoMLP.) *Given any $\epsilon > 0$ and any function $f \in C(\mathbb{X})$, if the function f is positive homogeneous as defined in Definition 1, there exist a finite-layer HomoMLP $\mathcal{A}'$, which represents the function $F_{\mathcal{A}'}$, such that for all $x \in \mathbb{X}$, we have $|F_{\mathcal{A}'}(x) - f(x)| < \epsilon$.*

We prove it based on Theorem A.2.2 by construction. In detail, according to Theorem A.2.2, there exists a finite-layer MLP $\mathcal{A}$ such that for all $x \in \mathbb{X}$, $|G_{\mathcal{A}}(x) - f(x)| = \left| |x| \, \mathcal{F}_{\mathcal{A}} \left(\frac{x}{|x|} \right) - f\left(x\right) \right| < \epsilon$. Without loss of generality, we assume

$\mathcal{A}$ as a two-layer ReLU feed-forward neural networks, which means $\mathcal{F}_{\mathcal{A}}(x) = W^2\text{ReLU}(W^1x + b^1) + b^2$, where $W^1 \in \mathbb{R}^{n\times m}, W^2 \in \mathbb{R}^{1\times n}$ are weight matrices and $b^1 \in \mathbb{R}^n, b^2 \in \mathbb{R}$ are biases. The function $G_{\mathcal{A}}$ can then be expressed as

$$G_{\mathcal{A}}(x) = |x|\left(W^2\text{ReLU}\left(W^1\frac{x}{|x|} + b^1\right) + b^2\right) = W^2\text{ReLU}\left(W^1x + b^1|x|\right) + b^2|x|. \tag{19}$$

Therefore, we just need to prove that the L1 norm $|\cdot|$ can be exactly calculated by HomoMLP to show that HomoMLP is a universal approximator homogeneous functions. Typically, we construct a two-layer HomoMLP as follows

$$\mathbf{1}^T\text{ReLU}\left(\begin{bmatrix}\mathbf{I}\\ -\mathbf{I}\end{bmatrix}x\right) \equiv |x|, \text{for all } x \in \mathbb{R}^m, \tag{20}$$

where $\mathbf{1}$ is a vertical vector containing $2m$ elements whose values are all one, and $\mathbf{I}$ denotes the identity matrix of size $m \times m$. Together with Eq. 19, we show that $G_{\mathcal{A}}(x)$ is a specification of HomoMLP, denoted as $\mathcal{A}'$, as follows:

$$\begin{aligned}
G_{\mathcal{A}}(x) &= W^2\text{ReLU}\left(W^1x + b^1|x|\right) + b^2|x| &(21)\\
&= W^2\text{ReLU}\left(W^1x + b^1\mathbf{1}^T\text{ReLU}\left(\begin{bmatrix}\mathbf{I}\\ -\mathbf{I}\end{bmatrix}x\right)\right) + b^2\mathbf{1}^T\text{ReLU}\left(\begin{bmatrix}\mathbf{I}\\ -\mathbf{I}\end{bmatrix}x\right) &(22)\\
&= \begin{bmatrix}W^2 & b^2\end{bmatrix}\text{ReLU}\left(\begin{bmatrix}W^1 + b^1 & -W^1 + b^1\\ \mathbf{1}^T & \mathbf{1}^T\end{bmatrix}\text{ReLU}\left(\begin{bmatrix}\mathbf{I}\\ -\mathbf{I}\end{bmatrix}x\right)\right) &(23)\\
&= F_{\mathcal{A}'}(x) &(24)
\end{aligned}$$

Here, we use $\mathbf{1}$ to denote a vertical vector containing m elements whose values are all one, and $\mathbf{I}$ still denotes the identity matrix of size $m \times m$. The summation of the weight matrix $W \in \mathbb{R}^{n\times m}$ and the bias $b \in \mathbb{R}^n$ outputs a new weight matrix W_b such that $W_b[i, j] = W[i, j] + b[i]$ for all $i = 1, 2, \cdots, n$ and $j = 1, 2, \cdots, m$, where $W[i, j]$ is the element in the ith row and the jth column of matrix W and $b[i]$ is the ith element of vertex b. Consequently, we build a three-layer HomoMLP $\mathcal{A}'$ with ReLU as activation functions such that for all $x \in \mathbb{X}$,

$$|F_{\mathcal{A}'}(x) - f(x)| = |G_{\mathcal{A}}(x) - f(x)| < \epsilon. \quad \square$$

B. Method

We present the decaying confidence mechanism to achieve much larger iteration numbers during inference in practice in Section B.1. We also show the formulation of PathGNN layers in detail in Section B.2. There are three variants of PathGNN, each of which corresponds to different degrees of flexibilities of approximating functions.

B.1. Decaying confidence mechanism of our iterative module

Although the vanilla IterGNN, as described in the main body, theoretically supports infinite iteration numbers, models can hardly generalize to much larger iteration numbers during inference in practice. In detail, we utilize the sigmoid function to ensure that confidence scores are between 0 and 1. However, the sigmoid function cannot predict zero confidence scores to continue iterations forever. Alternatively, the models will predict small confidence scores when they are not confident enough to terminate at the current time step. As a result, the models will still work well given the IID assumption, but cannot generalize well when much larger iteration numbers are needed than those met during training. For example, while solving the shortest path problem, 0.05 is a sufficiently small confidence score during training, because no iteration number larger than 30 is necessary and 0.95^{30} is still quite larger than 0. However, such models can not generalize to graphs with node numbers larger than 300, since $0.95^{300} \to 0$ and the models will terminate before time step 300 in any case. During our preliminary experiments, the vanilla IterGNN can not iterate for more than 100 times.

The key challenge is the difference between iteration numbers during training and inference. We then introduce a simple decaying mechanism to achieve larger iteration numbers during inference. The improved algorithm is shown in Algorithm 3. The termination probabilities will manually decay/decrease by λ at each time step. The final formulation of IterGNN can then generalize to iterate for 2500 times during inference in our experiments.

We compare the choices of decaying ratios as $0.99, 0.999, 0.9999$ in our preliminary experiments and fix it to 0.9999 afterwards in all experiments.

Algorithm 3 IterGNN with decay. g is the stopping criterion and f is the iteration body

Input: initial feature x; stop threshold ϵ; decay constant λ;
$k \leftarrow 1$
$h^0 \leftarrow x$
while $\lambda^k \prod_{i=1}^{k-1}(1-c^i) > \epsilon$ **do**
  $h^k \leftarrow f(h^{k-1})$
  $c^k \leftarrow g(h^k)$
  $k \leftarrow k+1$
end while
return $h = \lambda^k \sum_{j=1}^{k} c^j h^j \prod_{i=1}^{j-1}(1-c^i)$

B.2. Path Graph Neural Networks

As stated in (Battaglia et al., 2018), the performance of GNNs, especially their generalizability and zero-shot transferability, is largely influenced by the relational inductive biases. Xu (Xu et al., 2020) further formalized the relational inductive biases as sample efficiencies from algorithm alignments. For solving path-related graph problems such as the shortest path problem, a classical algorithm is the Bellman-Fold algorithm. Therefore, to achieve more effective and generalizable solvers for path-related graph problems, we design several specializations of GN blocks, as described in (Battaglia et al., 2018), by exploiting the inductive biases of the Bellman-Fold algorithms.

Our first observation of the Bellman-Fold algorithm is that it directly utilizes the input attributes such as the edge weights and the source/target node identifications at each iteration. We further observe that the input graph attributes of classical graph-related problems are all informative, well defined and also well represented as discrete one-hot encodings or simple real numbers (e.g. edge weights). Therefore, we directly concatenate the input node attributes with the hidden node attributes as the new node attributes before fed into our Graph Networks (GN) blocks. The models then do not need to extract and later embed the input graph attributes into the hidden representations in each GN block.

Our second observation is that the Bellman-Fold algorithm can be perfectly represented by Graph Networks as stated in Algorithm 4. Typically, for each iteration of the Bellman-Fold algorithm, the message module sums up the sender node's attributes (i.e. distance) with the edge weights, the node-level aggregation module then selects the minimum of all edge messages, and finally the attributes of the central node are updated if the new message (/distance) is smaller. The other modules of GN blocks are either identity functions or irrelevant. Therefore, to imitate the Bellman-Fold algorithm by GN blocks, we utilize the max pooling for both aggregation and update modules to imitate the min poolings in the Bellman-Fold algorithm. The edge message module is MLP, similarly to most GNN variants. The resulting module is then equal to replacing the MPNN (Gilmer et al., 2017)'s aggregation and update module with max poolings. Therefore, we call it MPNN-Max as in (Veličković et al., 2020). The concrete formulas are as follows

$$\begin{aligned} \mathbf{e}'_k &= MLP(\mathbf{v}_{s_k}, \mathbf{v}_{r_k}, \mathbf{e}_k) \\ \bar{\mathbf{e}}_j &= \max(\{\mathbf{e}'_k : r_k = j\}) \\ \mathbf{v}'_j &= \max(\mathbf{v}_j, \bar{\mathbf{e}}_j) \end{aligned}$$

Moreover, we notice that the Bellman-Ford algorithm is only designed for solving the shortest path problem. Many path-related graph problems can not be solved by it. Therefore, we further relax the max pooling to attentional poolings to increase the models' flexibility while still maintaining the ability to approximate min pooling in a sample efficient way. Typically, we propose the PathGNN by replacing the aggregation module with attentional pooling. The detailed algorithm is stated in Algorithm 5, where the global attributes are omitted due to their irrelevance.

Another variant of PathGNN is also designed by exploiting a less significant inductive bias of the Bellman-Fold algorithm. Specifically, we observe that only the sender node's attributes and the edge attributes are useful in the message module while approximating the Bellman-Fold algorithm. Therefore, we only feed those attributes into the message module of our new PathGNN variant, PathGNN-sim. The detailed algorithm is stated in Algorithm 7.

Algorithm 4 The Bellman-Fold algorithm

Input: node attributes $V = \{\mathbf{v}_i, i = 1, 2, \cdots, N_v\}$, edge attributes $E = \{(w_k, s_k, r_k), k = 1, 2, \cdots, N_e\}$, and the source node $source$.
Output: The shortest path length from the source node to others, $distance = \{dist_i, i = 1, 2, \cdots, N_v\}$.

Initialize the intermediate node attributes $V' = \{dist_i, i = 1, 2, \cdots, N_v\}$ as $dist_i = \begin{cases} 0 & if.\ i = source \\ \infty & o.w. \end{cases}$.
for i=1 **to** $N_v - 1$ **do**
  for (w_k, s_k, r_k) **in** E **do**
    $\mathbf{e}'_k = dist_{s_k} + w_k$ ◁ edge message module
  end for
  for j=1 **to** N_v **do**
    $\bar{\mathbf{e}}_j = \min(\{\mathbf{e}'_k : r_k = j\})$ ◁ aggregation module
    $dist_j = \min(dist_j, \bar{\mathbf{e}}_j)$ ◁ update module
  end for
end for

Algorithm 5 One step of PathGNN

Input: graph $G = (V, E)$
Output: updated graph $G' = (V', E)$

for (w_k, s_k, r_k) **in** E **do**
  $\tilde{\mathbf{e}}_k = MLP(\mathbf{v}_{s_k}, \mathbf{v}_{r_k}, \mathbf{e}_k)$
  $score_k = MLP'(\mathbf{v}_{s_k}, \mathbf{v}_{r_k}, \mathbf{e}_k)$
  $\mathbf{e}'_k = (score_k, \tilde{\mathbf{e}}_k)$ ◁ edge message module
end for
for j=1 **to** N_v **do**
  $\bar{\mathbf{e}}_j = \text{attention}(\{\mathbf{e}'_k : r_k = j\})$ ◁ aggregation module
  $\mathbf{v}'_j = \max(\mathbf{v}_j, \bar{\mathbf{e}}_j)$ ◁ update module
end for

In summary, we introduce Path Graph Neural Networks (PathGNN) to improve the generalizability of GNNs for distance related problems by improving the algorithm alignment (Xu et al., 2020). It is a specially designed GNN layer that imitates one iteration of the classical Bellman-Ford algorithm. There are three variants of PathGNN, i.e. MPNN-Max, PathGNN, and PathGNN-sim, each of which corresponds to different degrees of flexibilities. In our experiments, they perform much better than GCN and GAT for all path-related tasks regarding the generalizability, as stated in Section 5 in the main body.

C. Experiment Setups

We present the detailed experimental setups in this subsection. We first present details of three graph theory problems, i.e. the shortest path problem in Section C.1.2, the component counting problem in Section C.1.3, and the traveler salesman problem in Section C.1.4. We then list the experimental setups of three graph-based reasoning tasks, i.e. the physical simulation in Section C.2.1, the image-based navigation in Section C.2.2, and the symbolic Pacman in Section C.2.3. We also describe the

Algorithm 6 Attention Pooling in GNNs

Input: set of messages $\{\mathbf{e}'_k = (score_k, \tilde{\mathbf{e}}_k)\}$
Output: aggregated messages $\bar{\mathbf{e}}_j$

$\alpha = softmax(score)$
$\bar{\mathbf{e}}_j = \sum_k \alpha_k \tilde{\mathbf{e}}_k$

Algorithm 7 One step of PathGNN-sim

Input: graph $G = (V, E)$
Output: updated graph $G' = (V', E)$

for (w_k, s_k, r_k) **in** E **do**
  $\tilde{\mathbf{e}}_k = MLP(\mathbf{v}_{s_k}, \mathbf{e}_k)$
  $score_k = MLP'(\mathbf{v}_{s_k}, \mathbf{v}_{r_k}, \mathbf{e}_k)$
  $\mathbf{e}'_k = (score_k, \tilde{\mathbf{e}}_k)$ ◁ edge message module
end for
for j=1 **to** N_v **do**
  $\bar{\mathbf{e}}_j = \text{attention}(\{\mathbf{e}'_k : r_k = j\})$ ◁ aggregation module
  $\mathbf{v}'_j = \max(\mathbf{v}_j, \bar{\mathbf{e}}_j)$ ◁ update module
end for

experimental setups of graph classification in Section C.3. The descriptions of models are stated in Section C.4 and the training details are presented in Section C.5.

C.1. Graph theory problems

We evaluate our proposals on three classical graph theory problems, i.e. shortest path, component counting, and the Traveling Salesman Problem (TSP). We present the properties of graph generators in Section C.1.1, the setups for the shortest path problem in Section C.1.2, the setups for the component counting in Section C.1.3, and the setups for the TSP problem in Section C.1.4.

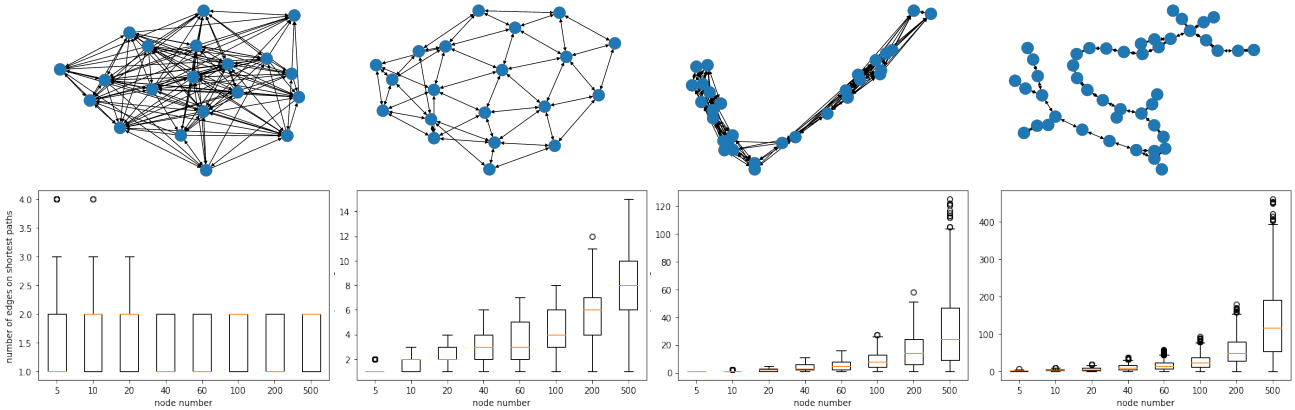
C.1.1. Properties of Graph Generators

How to sample graphs turns to be intricate in our exploration to deeply investigate the generalizability power. Four generators are adopted in our experiments:

- The Erdos-Renyi model (Erdos & Renyi, 1959), $G(n, p)$, generates graphs with n nodes and each pair of nodes is connected with probability p.
- The KNN model, KNN(n, d, k), first generates n nodes whose positions are uniformly sampled from a d-dimensional unit cube. The nodes are then connected to their k nearest neighbors. The edge directions are from the center node to its neighborhoods.
- The planar model, PL(n, d), first generates n nodes whose positions are uniformly sampled from a d-dimensional unit cube. Delaunay triangulations are then computed (Barber et al., 1996) and nodes in the same triangulation are connected to each other.
- The lobster model, Lob(n, p_1, p_2), first generates a line with n nodes. $n_1 \sim \mathcal{B}(n, p_1)$ nodes are then generated as first-level leaf nodes, where $\mathcal{B}$ denotes the binomial distribution. Each leaf node uniformly selects one node in the line as the parent. Afterwards, $n_2 \sim \mathcal{B}(n_1, p_2)$ are generated as second-level leaf nodes. Each second-level leaf node also uniformly selects one first-level leaf node as the parent. The parents and children are connected to each other and the graph is therefore undirected.

For the Erdos-Renyi model, we assign p equal to 0.5 so that all graphs of n nodes can be generated with equal probabilities. However, the expectation of graph diameters decreases dramatically as graph sizes increase for such model. As illustrated in Figure 3, the graph diameters are just 2 with high probability when the node number is larger than 50.

The other three graph generators are therefore designed to generate graphs with larger diameters for better evaluation of models' generalizabilities w.r.t. scales. We manually select their hyper-parameters to efficiently generate graphs of diameters as large as possible. Specifically, we set $d = 1$ and $k = 8$ for the KNN model, $d = 2$ for the planar model, $p_1 = 0.2$ and $p_2 = 0.2$ for the lobster model. Their properties are illustrated in Figure 3. Note that the distances are positively related to the graph sizes for all three graph generators. Moreover, for the lobster model, the distances increase almost linearly with respect to the graph sizes.



(a) Erdős–Rényi model (b) Planar graph (dim=2) (c) KNN graph (dim=1, k=8) (d) Lobster graph (p1=p2=0.2)

Figure 3. Properties of different random graph generators. The upper row illustrates graph samples generated by the corresponding generator. The lower row demonstrates the relationships between the graph sizes (i.e. node numbers) and the distributions of random node pairs' distances. Typically, for each generator and each graph size, 1000 sample graphs are generated by the corresponding generator for estimating the distributions. Box plots are utilized for visualizing the distributions.

C.1.2. SHORTEST PATH

In this task, given a source node and a target node, the model needs to predict the length of the shortest path between them. The edge weights are positive and uniformly sampled. We consider both unweighted graphs and weighted graphs (edge weights uniformly sampled between 0.5 and 1.5). Groundtruth is calculated by Dijkstra's algorithm.

We utilize a three-dimensional one-hot representation to encode the location of the source and target nodes (100 for source, 010 for destination, and 001 for other nodes). Edge weights are encoded as the edge attributes. Two metrics are used to measure the performance of GNNs. The relative loss is first applied to measure the performance of predicting shortest path lengths. To further examine the models' ability in tracing the shortest path, we implemented one simple post-processing method leveraging the noisy approximation of path lengths (presented later). In detail, we define the relative loss as $|l - l_{pred}|/l$ and the success rate of identifying the shortest path by post-processing as $\mathbb{1}(l = l_{post\text{-}pred})$, where l is the true shortest path length, l_{pred} is the predicted length by GNNs, and $l_{post\text{-}pred}$ is the length of the predicted path after post-processing.

Post-Processing After training, our model can predict the shortest path length from source to target nodes. The post-processing method is then applied to find the shortest path based on the learned models. Specifically, the post-processing method predicts the shortest path

$$p = [p_1, p_2, \cdots, p_n] = \text{post-processing}(G; GNN)$$

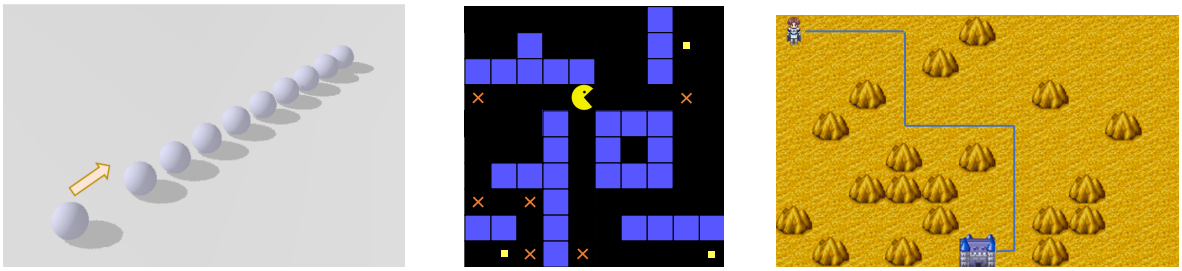
given the input graph $G = (V_{i,j}, E)$, as defined in Section 5.1.1., and the noisy shortest path length predicting model GNN, where i, j are the indexes of source and target nodes, $V_{i,j}$ is the node attributes with one-hot encodings, and p_k denotes the index of the kth node on the shortest path.

The post-processing algorithm is stated in Algorithm.8. Concretely, we first denote the shortest path length between any two nodes i' and j' predicted by the trained model GNN as $dist_{i',j'} = GNN(G_{i',j'}) = GNN((V_{i',j'}, E))$, where $V_{i',j'}$ represents the one-hot node attributes when nodes i' and j' are the source and target nodes. For further convenience, we also defined $w_{i',j'}$ as the weight of edge connecting node i' and node j' ($w_{i',j'} = \infty$ if node i' and node j' are not connected by an edge.). Then, the post-processing algorithm sequentially predicts the next node p_{k+1} of node p_k by minimizing the difference between the predicted shortest path length approximated by GNNs $dist_{p_k,j}$ and the length of shortest path predicted by the post-processing method $dist_{p_{k+1},j} + w_{p_k,p_{k+1}}$. To further reduce the effect of models' noises, another constrain is added as $dist_{p_{k+1},j} + w_{p_k,p_{k+1}} \leq dist_{p_k,j}$ so that the method will always convergence.

Algorithm 8 Post-processing to predict the shortest path

Input: graph $G = (V_{i,j}, E)$; source node index i; target node index j; trained models GNN that predict the shortest path length from nodes i' to node j' as $dist_{i',j'} = GNN((V_{i',j'}, E))$
Output: shortest path $p = [p_1, p_2, \cdots, p_n]$

Initialize $p_1 = i, k = 1$.
while $p_k \neq j$ **do**
  if $|\{l : dist_{l,j} + w_{p_k,l} \leq dist_{p_k,j}\}| > 0$ **then**
    $p_{k+1} = \arg\min_{l:dist_{l,j}+w_{p_k,l} \leq dist_{p_k,j}} |dist_{l,j} + w_{p_k,l} - dist_{p_k,j}|$.
  else
    return $p = []$
  end if
  $k = k + 1$.
end while
return p



(a) Newton's ball (b) Symbolic PacMan (c) Image-based Navigation

Figure 4. Figure (a) shows a set of Newton's balls in the physical simulator. The yellow arrow shows the moving direction of the first ball. Figure (b) is a scene in our symbolic PacMan environment. Figure (c) illustrates our image-based navigation task in a RPG-game environment.

C.1.3. Component Counting

In component counting task, the model counts the number of connected components of an undirected graph. To generate a graph with multiple components, we first sample a random integer m between 1 to 6 as the number of components, and then divide the nodes into m parts. In detail, for n nodes and m components, we first uniformly sample $m - 1$ positions from 1 to $n - 1$ and then divide the n nodes into m parts by the $m - 1$ positions. We then connect nodes in each component using the random graph generator defined above, e.g., Erdos-Renyi graph and lobster graph. The metric is the accuracy of correct counting. We initialize the node attributes by random values$\in [0, 1)$ to distinguish different nodes.

C.1.4. Traveler Salesman Problem (TSP)

In the Euclidean travelling salesman problem (TSP), there are several 2D points located in the Euclidean plane, and the model generates the shortest route to visit each point. The graph is complete. The weight of an edge is the Euclidean distance between the two ends. Points $\{(x_i, y_i)\}$ are uniformly sampled from $\{x, y \in \mathbb{Z} : 1 \leq x, y \leq 1000\}$. We use the standard solver for TSP, Concorde (Applegate et al.) to calculate the ground truth. The node attributes are the 2D coordinates of each node. We use relative loss defined the same as the shortest path problem to evaluate the networks.

C.2. Graph-based reasoning tasks

We further evaluate the benefits of our proposals using three graph-based reasoning tasks. We describe the setups of physical simulation in Section C.2.1, the setups of symbolic Pacman in Section C.2.3, and the setups of image-based navigation in Section C.2.2.

C.2.1. PHYSICAL SIMULATION

We evaluate the generalizability of our models by predicting the moving patterns between objects in a physical simulator. We consider an environment similar to *Newton's cradle*, also known as called *Newton's ball*, as shown in Figure 4(a): all balls with the same mass lie on a friction-free pathway. With the ball at one end moving towards others, our model needs to predict the motion of the balls of both ends at the next time step. The probability is 50% for balls to collide. We represent the environment as a chain graph. The nodes of the graph stand for the balls and the edges of the graph stand for the interactions between two adjacent balls. We fix the number of balls within $[4, 34)$ at the training phase, while test the networks in environments with 100 nodes.

In detail, we generate samples as follows:

- $n-1$ balls with same properties are placed as a chain in the one-dimensional space where each ball touches its neighbourhoods.
- A new ball moves towards the $n-1$ balls from left position x with constant speed v.
- The model needs to predict each ball's position and speed after one time step.

The radius of balls, r, is set to 0.1 and the positions are normalized so that the origin is in the middle of $n-1$ balls. The left ball's position x is set so that its distance to the most left ball among the other $n-1$ balls is uniformly sampled between 0 and $200r$. The left ball's speed v is uniformly sampled between 0 and $200r$. Note that the left ball may or may not collide with the other balls depending on its positions and weights. The probability is 50% for balls to collide.

Since the $n-2$ balls in the middle will not change their positions or speeds in any cases, we simplify the output to the positions and speeds of the left and right balls. To avoid trivial solutions, we still force the models to predict positions and speeds at the node level, which means no global readout modules are allowable.

C.2.2. IMAGE-BASED NAVIGATION

We show benefits of the differentiability of a generalizable reasoning module using the image-based navigation task, as illustrated in Figure 4(c). The model needs to plan the shortest route from source to target on 2D images with obstacles. However, the properties of obstacles are not given as a prior and the model must discover them based on image patterns during training.

We simplify the task by defining each pixel as obstacles merely according to its own pixel values. Specially, we assign random heights from $[0, 1)$ to pixels in 2D images. The agent cannot go through pixels with heights larger than 0.8 during navigation. The cases where no path exists between the source node and the target node are abandoned. We represent the image by a grid graph. Each node corresponds to a pixel. The edge attributes are set to one. The node attributes are the concatenation of pixel values and the one-hot embedding of node's categories (source/target/others). Note that, for more complex tasks, the node attributes can also include features extracted by CNNs.

In detail, we generate samples as follows:

- $n \times n$ grid is first generated. Each node is connected to its left, right, up, and bottom neighbourhoods. (The boundary situations are omitted for simplicity)
- The height, h_i, is uniformly sampled between 0 and 1, and is then assigned to node i. Nodes with heights larger than 0.8 cannot be visited.
- The source node and the target node are uniformly sampled from node pairs that have heights less than 0.8 and are connected.

The node attributes are their heights and the one-hot encodings of their categories (i.e. source, target, or others). The edge attributes are all ones. The properties of datasets are visualized in Figure 5.

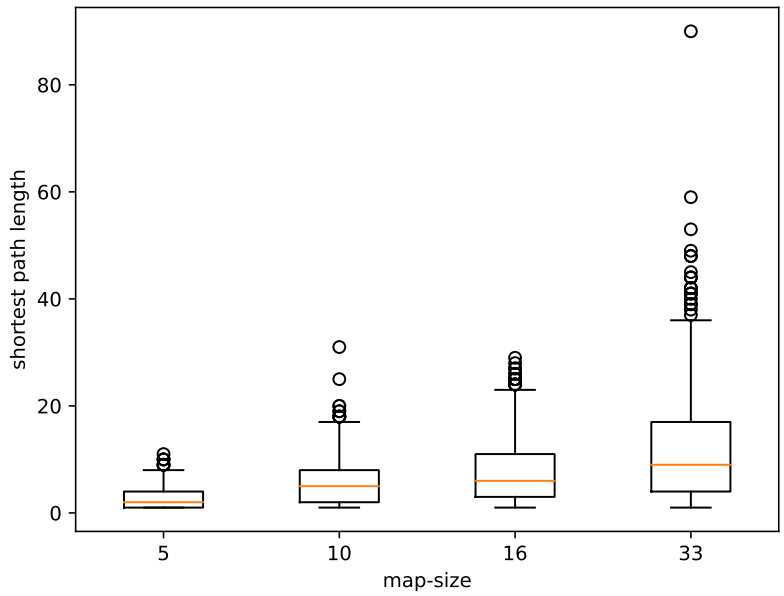


Figure 5. Illustration of the properties of the datasets for image-based navigation. Box plots are utilized to visualize the distributions of the lengths of the shortest path between two random nodes for each map-size. The shortest path lengths increase with the map sizes.

C.2.3. SYMBOLIC PACMAN

To show that our iterative module can improve reinforcement learning, we construct a symbolic PacMan environment with similar rules to the PacMan in Atari (Bellemare et al., 2013). As shown in Figure 4(b), the environment contains a map with dots and walls. The agent starts at one position and at each step it can move to one of neighboring grids. The agent will receive a reward of 1 when it reaches one dot and "eats" the dot. The discount factor is set to 0.9. The agent needs to figure out a policy to quickly "eat" all dots while avoiding walls on the map to maximize the return.

In detail, we generate random environments as follows:

- Maze of size 16×16 is first generated.
- n_w walls of length 3 are then generated.
- The walls' directions are assigned randomly as vertical or horizontal.
- The walls' positions are uniformly sampled from all feasible ones regardless of overlappings.
- n_d dots are further generated with positions uniformly sampled from positions that are not occupied by walls.
- one agent is at last generated with positions uniformly sampled from non-occupied ones.

The model then controls the agent to navigate in the maze to eat all dots. The action space is [left, right, up, down]. The agent will not move if the action is infeasible such as colliding with the walls. The game will stop if the agent has eaten all dots or has exceeded the maximum time step, which is $16 \times 16 \times (n_d + 1)$. A new environment will be generated afterwards. The metric is then the success rates of eating dots:

$$\frac{\text{number of eaten dots}}{\text{number of reachable dots}}$$

At each step, landmarks are placed on each corner of the shortest paths from the agent to dots. The input state is then a graph with the agent, dots, and landmarks as nodes. The node attributes are their positions plus one-hot encodings of their

categories. The positions are normalized so that the agent is at the origin. The edge weights are set as the Manhattan distance between every two nodes.

We train our models using the double DQN (Van Hasselt et al., 2016) with value networks replaced by our backbones. The reward for eating each dot is 1 and no penalty for colliding with walls. The discount is set to 0.9 for each time step to encourage faster navigation. The exploration probability is 0.1 and the warm-up exploration steps are 1000. Value networks are trained every 4 time steps and updated every 200 time steps. The size of the replay buffer is 10000. The batch size is 32 and the learning rate is 0.0002. Models are tested on 200 different environments and the averaged performance is reported.

The network architectures are as follows: 2-layer MLP with leaky ReLU for feature embedding, GNN/CNN modules for message passing, max pooling for readout, and 1-layer FC for predicting the Q values. The hidden sizes are 64. We compare our IterGNNs with PointNet (Qi et al., 2017), GCN (Kipf & Welling, 2017), and CNNs. For the PointNet model, the GNN modules are identities by definition. For the GCN and CNN models, we compare the performance of their 1/3/5/7/9-layer variants and report the best of them. We also compare the choice of the kernel sizes of CNNs among 3/5/7 and report the best of them.

C.3. Graph Classification

Note that models' abilities to utilize the information of long-term relationships are necessary for accurately solving most of the previous tasks and problems. Therefore, the benefits of adaptive and unbounded depths introduced by the iterative module are distinguished. In this sub-section, we show that our IterGNN can also achieve competitive performance on graph-classification benchmarks, demonstrating that our iterative module does not hurt the standard generalizability of GNNs while improving the generalizability w.r.t. graph scales. The results are presented in Section D.3.

In detail, we evaluate models on five small datasets, which are two social network datasets (IMDB-B and IMDB-M) and three bioinformatics datasets (MUTAG, PROTEINS and PTC). Readers are referred to (Xu et al., 2019) for more descriptions of the properties of datasets. We adopt the same evaluation method and metrics as previous state-of-art (Xu et al., 2019), such as 10-fold cross-validation and classification accuracies as the metric. The dataset splitting strategy and pre-processing methods are all identical to (Xu et al., 2019) by directly integrating their public codes[1].

Regarding the models, we adopt the previous state-of-art, GIN (Xu et al., 2019) as GN-blocks and the JK connections (Xu et al., 2018) plus average/max pooling as the readout module. To integrate the iterative architecture, we wrap each GN-block in original backbones with the iterative module with maximum iteration number equal to 10. The tunable hyper-parameters include the number of IterGNNs, the epoch numbers, and whether or not utilizing the random initialization of node attributes.

C.4. Models and Baselines

We follow the common practice of designing GNN models as presented in Section F.2. We utilize a 2-layer MLP for node attribute embedding and use a 1-layer MLP for prediction. The max/sum poolings are adopted as readout functions to summarize information of a graph into one vector.

To build the core GNN module, we need to specify three properties of GNNs: the GNN layers, the paradigms to compose GN-blocks, and the prior, as stated in Section F.2. In our experiments, we explore the following options for each property:

- GNN layers: PathGNN layers and two baselines that are GCN (Kipf & Welling, 2017) and GAT (Veličković et al., 2018).
- Prior: whether or not apply the homogeneous prior
- Paradigm to compose GN-blocks: our iterative module; the simplest paradigm that stacks multiple GNN layers sequentially; the ACT algorithm (Graves, 2016); and the fixed-depth shared-weights paradigm, as described in the main body.

For the homogeneous prior, we apply the prior to the node-wise embedding module, the readout module, and the final prediction module as well for most problems and tasks. However, for problems whose solutions are not homogeneous (e.g. component counting), we only apply the homogeneous prior to the core GNN module.

[1] https://github.com/weihua916/powerful-gnns

In detail, models are specified by the choices of the previous properties. The corresponding models and their short names are as follows:

- *GCN, GAT*: Models utilizing the multi-layer architecture (i.e. stacking multiple GN-blocks) with GCN (Kipf & Welling, 2017) or GAT (Veličković et al., 2018) as GN-blocks. The homogeneous prior is not applied.
- *Path / Multi-Path*: Models utilizing the multi-layer architecture and adopting PathGNN as defined in Section B.2 as GN-blocks. No homogeneous prior is applied.
- *Homo-Path / Multi-Homo-Path*: Models utilizing the multi-layer architecture and adopting PathGNN as GN-blocks. And the homogeneous prior is applied on all modules unless otherwise specified.
- *Iter-Homo-Path / Iter-HP*: Models utilizing the iterative module as described in Section B.1 and adopting PathGNN as GN-blocks. The homogeneous prior is applied on all modules unless otherwise specified.
- *Shared-Homo-Path, ACT-Homo-Path*: Models utilizing the fixed-depth and shared-weights paradigm (i.e. repeating one GN-block for multiple times) and the adaptive computation time algorithm (Graves, 2016), respectively. PathGNN and the homogeneous prior are all applied.
- *Iter-Path*: Same as Iter-Homo-Path except that no homogeneous prior is applied.
- *Iter-GAT*: Models utilizing our iterative module and adopting GAT as GN-blocks. No homogeneous prior is applied.

For most problems, max pooling is utilized as the readout function and we use only one IterGNN to build the core graph neural networks. The homogeneous prior is applied to all modules. However, for component counting, sum pooling is utilized as the readout function and two IterGNNs are stacked sequentially, since two iteration loops are usually required for component-counting algorithms (one for component assignment and one for counting). We utilize the node-wise IterGNN to support unconnected graphs. The homogeneous prior is only applied to the GNN modules but not the embedding module and count prediction module, because the problem is not homogeneous. Random initialization of node attributes is also applied to improve GNNs' representation powers.

C.5. Training Details

All models are trained with the same set of hyper-parameters: the learning rate is 0.001, and the batch size is 32. We use Adam as the optimizer. The hidden neuron number is 64. For models using the iterative module or the ACT algorithm, we train the networks with 30 maximal iterations and test them with no additional constraints. For the fixed-depth shared-weights paradigm, we train the networks with 30 iterations and test them with 1000 iterations (maximum node numbers in the datasets). The only two tunable hyper-parameter within our proposals are the epoch number=20, 40, $\cdots$, 200 and the degree of flexibilities of PathGNN, each corresponding to one variation of the PathGNN layer as described in Section B.2. Another hyper-parameter within the ACT algorithm (Graves, 2016) is $\tau = 0, 0.1, 0.01, 0.001$. We utilize the validation dataset to select the best hyper-parameter and report its performance on the test datasets.

D. Experiment Results

D.1. Solving graph theory problems

D.1.1. Ablation studies and comparison

We conduct ablation studies to exhibit benefits of our proposals using the unweighted shortest path problem on lobster graphs with 1000 nodes in Table 2. The models are built by replacing each proposal in our best Iter-Homo-Path model with other possible substitutes in the literature. For the iterative module, other than the simplest paradigm utilized in Homo-Path that stacks GNN layers sequentially, we also compare it with the ACT algorithm (Graves, 2016) and the fixed-depth weight-sharing paradigm (Tamar et al., 2016; Li & Zemel, 2014), resulting in the "ACT-Homo-Path" and "Shared-Homo-Path" models. The ACT algorithm provides adaptive but usually small iterations of layers. The weight-sharing paradigm iterates modules for predefined times and assumes that the predefined iteration number is large enough. We set its iteration number to the largest graph size in the dataset. Homo-Path and ACT-Homo-Path perform much worse than Iter/Shared-Homo-Path because of the limited representation powers of shallow GNNs. Shared-Homo-Path performs worse than our Iter-Homo-Path,

Iter-Homo-Path	
100.0	
Homo-Path	Iter-Path
53.7	48.9
ACT-Homo-Path	Iter-Homo-GAT
52.7	2.9
Shared-Homo-Path	Iter-Homo-GCN
91.7	1.4

Table 2. Ablation studeis of generalization performance for the shortest path problem on lobster graphs with 1000 nodes. Metric is the success rate.

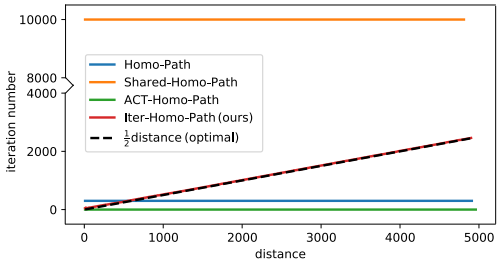


Figure 6. The iteration numbers of GNN layers w.r.t. the distances from the source node to the target node for the shortest path problem.

Table 3. Generalization performance on graph-related reasoning. Models are trained on graphs of smaller sizes (e.g. within $[4, 34)$ or $\leq 10 \times 10$) and are tested on graph of larger sizes (e.g. 50, 100, 16×16 or 33×33). The metric for physical simulation is the mean square error. The metric for image-based navigation is the success rate.

Models	Physical sim. 50	Physical sim. 100	Image-based Navi. 16×16	Image-based Navi. 33×33
GCN (Kipf & Welling, 2017)	42.18	121.14	34.2%	28.9%
GAT (Veličković et al., 2018)	>1e4	>1e4	56.7%	44.5%
Path (ours)	20.24	27.67	85.6%	65.1%
Homo-Path (ours)	20.48	21.45	87.8%	84.4%
Iter-Homo-Path (ours)	**0.12**	**2.01**	**98.8%**	**91.7%**

possibly because of the accumulated errors after unnecessary iterations. For the homogeneous prior, we build "Iter-Path" by simply removing the homogeneous prior. It performs much worse than Iter-Homo-Path because of the poor performance of MLPs on out-of-distribution features. For PathGNN, we build "Iter-Homo-GCN" and "Iter-Homo-GAT" by replacing PathGNN with GCN and GAT. Their bad performance verifies the benefits of better algorithm alignments (Xu et al., 2020).

D.1.2. Interpreting stopping criterion learned by the iterative module.

We show that our Iter-Homo-Path model learned the optimal stopping criterion for the unweighted shortest path problem in Figure 6. Typically, to accurately predict the shortest path of lengths d on undirected graphs, the iteration number of GNN layers is at least $d/2$ due to the message passing nature of GNNs. In detail, both the source node and the target node need to send and to collect information along the shortest path towards each other, so that each iteration of GNNs could collect information of two more edges on the shortest path. No fewer iteration numbers can be achieved using 1-hop message passing GNNs as the path length is already the shortest. Our iterative module actually learns such optimal stopping criterion. The Iter-Homo-Path model adaptively increases the iteration numbers w.r.t. the distances and moreover stops timely when the information is enough.

D.2. General reasoning tasks

D.2.1. Physical simulation.

We evaluate the generalizability of our models by predicting the moving patterns between objects in a physical simulator. We consider an environment called *Newton's ball*: all balls with the same mass lie on a friction-free pathway. With the ball at one end moving towards others, our model needs to predict the motion of the balls of both ends at the next time step. The metric is the mean squared error. Models are trained in worlds with $[4, 34)$ balls and are tested in worlds with 100 balls. As shown in Table 3, the Iter-Homo-Path model significantly outperforms others demonstrating the advantages of our iterative module for improving generalizability w.r.t. scales. The homogeneous prior is beneficial even though the target functions are not homogeneous.

Table 4. Generalization performance of IterGNN for symbolic Pacman. Metric is the success rate of eating dots. Models are trained in environments with 10 dots and 8 walls and are tested in environments with different number of walls and dots.

#wall \ #dots	1	5	10	15	20
0	1.00	1.00	0.99	0.99	1.00
3	0.95	1.00	0.98	0.94	0.98
6	0.90	0.95	0.94	0.97	0.98
9	0.80	0.92	0.95	0.95	0.93
12	0.60	0.96	0.97	0.98	0.93
15	0.75	0.92	0.94	0.95	0.97

Table 5. Generalization performance of GCN for symbolic Pacman. Metric is the success rate of eating dots. Models are trained in environments with 10 dots and 8 walls and are tested in environments with different number of walls and dots.

#wall \ #dots	1	5	10	15	20
0	0.05	0.23	0.09	0.07	0.05
3	0.10	0.23	0.20	0.15	0.04
6	0.00	0.32	0.17	0.09	0.06
9	0.20	0.27	0.23	0.06	0.10
12	0.05	0.18	0.26	0.13	0.03
15	0.05	0.17	0.23	0.15	0.08

D.2.2. SYMBOLIC PACMAN

To show that our iterative module can improve reinforcement learning, we construct a symbolic PacMan environment with similar rules to the PacMan in Atari (Bellemare et al., 2013). The environment contains a map with dots and walls. The agent needs to figure out a policy to quickly "eat" all dots while avoiding walls on the map to maximize the return. We abstract observations as graphs using the landmark (Huang et al., 2019). We adopt Double Q learning (Van Hasselt et al., 2016) to train the policy. Unlike original Atari PacMan, our environment is more challenging because we randomly sample the layout of maps for each episode and we test models in environments with different numbers of dots and walls. The agent can not just remember one policy to get successful but needs to learn to do planning according to the current observation. The metric is the success rate of eating dots. The experimental setups are presented in Section C.2.3.

Table 4, Table 5, and Table 6 show the performance of IterGNN, GCN (Kipf & Welling, 2017) and PointNet (Qi et al., 2017), respectively, in environments with different number of walls and dots. Our IterGNN demonstrates remarkable generalizability among different environment settings, as stated in Table 4. It successfully transfers policies to environments with different number of dots and different number of walls. IterGNN performs much better than the GCN and PointNet baselines, demonstrating that our proposals improve the generalizability of models. GCN performs the worst probably because of the unsuitable strong inductive bias encoded by the normed-mean aggregation module.

Table 6. Generalization performance of PointNet for symbolic Pacman. Metric is the success rate of eating dots. Models are trained in environments with 10 dots and 8 walls and are tested in environments with different number of walls and dots.

#wall \ #dots	1	3	6	9	12	15
2	0.82	0.58	0.39	0.32	0.34	0.29
4	0.72	0.48	0.31	0.25	0.24	0.22
6	0.71	0.31	0.36	0.24	0.19	0.14
8	0.60	0.36	0.21	0.20	0.20	0.28
10	0.50	0.33	0.33	0.29	0.16	0.15

Table 7. The performance of our iterative module on graph classification benchmarks. Iter-GIN is built by wrapping each GNN layer in the previous state-of-art GIN model using our iterative module. Metric is the averaged accuracy and STD in 10-fold cross-validation.

Dataset	GIN (Xu et al., 2019)	Iter-GIN (ours)
IMDB-B	75.1±5.1	**75.7±4.2**
IMDB-M	**52.3±2.8**	51.8±4.0
MUTAG	89.4±5.6	**89.6±8.6**
PROTEINS	76.2±2.8	**76.3±3.4**
PTC	**64.6±7.0**	64.5±3.8

D.2.3. IMAGE-BASED NAVIGATION.

We show benefits of the differentiability of a generalizable reasoning module using the image-based navigation task. The model needs to plan the shortest route from source to target on 2D images with obstacles. However, the properties of obstacles are not given as a prior and the model must discover them based on image patterns during training. We simplify the task by defining each pixel as obstacles merely according to its own pixel values. As stated in Table 3, our Iter-Homo-Path model successfully solves the task. The model achieves success rates larger than 90% for finding the shortest paths on images of size 16×16, and 33×33, while is only trained on images of size $\leq 10 \times 10$. All of our proposals help improve the generalizability.

D.3. Graph Classification

We evaluate models on standard graph classification benchmarks to show that our proposals do not hurt GNNs' powers as well as the standard generalizability while improving their generalizability w.r.t. scales. More descriptions of the task are available in (Xu et al., 2019). The experimental setups are presented in Section C.3.

As stated in Table 7, our model performs competitively with the previous state-of-art backbone, GIN, on all five benchmarks. It suggests that our iterative module is a safe choice for improving generalizability w.r.t. scales while still maintaining the performance for normal tasks. Note that, due to the shortage of time, little hyper-parameter search was conducted in our experiments. Default hyper-parameters such as learning rate equal to 0.001 and hidden size equal to 64 were adopted. Therefore, the performance of Iter-GIN is potentially better than those stated in Table 7.

E. Related Work

Graph Neural Networks. Graph neural networks (GNNs) have become popular in many fields (Zhou et al., 2018; Goyal & Ferrara, 2018; Wu et al., 2020) , because of their abilities to handle irregular graph structures, to approximate permutation invariant/equivalent functions, and to exploit relational inductive biases such as locality (Battaglia et al., 2018). Most of those popular GNN variants come within a generalized framework called message passing (Gilmer et al., 2017; Battaglia et al., 2018). We will describe it in Section 3.

Graph Algorithm Learning. Despite the success of GNNs (mostly come within the message passing framework (Gilmer et al., 2017; Battaglia et al., 2018)) in many fields (Zhou et al., 2018; Goyal & Ferrara, 2018; Wu et al., 2020), few works have reported remarkable results on solving traditional graph-related problems, such as the shortest path problem, by neural networks, especially when the generalizability with respect to scales is taken into account. Previously, in the period with limited computation power, people explored the possibility of approximating graph algorithms by neural networks for more efficiencies by distributed computation (Araujo et al., 2001). After the resurgence of deep neural networks, Neural Turing Machine (Graves et al., 2014; 2016) first reported performance on solving shortest path on small graphs using deep neural networks to demonstrate its representation powers. Recently, (Veličković et al., 2020) and (Xu et al., 2020) achieved positive performance on solving the shortest path problem on relatively large graphs using the GNNs. However, methods in (Veličković et al., 2020) require per-layer supervisions to train and models in (Xu et al., 2020) theoretically lack the generalizability w.r.t. graph scales due to their bounded number of message passing steps. As far as we know, no previous work has solved the shortest path problem by neural networks on graphs of diameters larger than 100. There is either no work reporting notable generalization performance with respect to graph scales.

Iterative Algorithm Approximation. Traditional iterative algorithms have been successfully applied in solving the classical graph problems (Dijkstra, 1959) as well as in many application fields such as controlling, network analysis (Page et al.,

1999), and Bayesian statistics (Blei et al., 2017). Inspired by their success, several works were proposed to incorporate the iterative architecture into neural networks for better generalizability (Tamar et al., 2016), more efficiency (Dai et al., 2018), or to support end-to-end training (Li & Zemel, 2014; Zheng et al., 2015). However, none of them supports adaptive and unbounded iteration numbers and is therefore not applicable for approximating general iterative algorithms over graphs of any sizes. Dai (Dai et al., 2018) adopted GNNs to improve the efficiency and scalability of a special class of iterative algorithms whose solutions are characterized by a set of steady-state conditions. However, it does not apply to approximate general iterative algorithms and the iteration numbers are usually pre-defined as done in their experiments.

Differentiable Controlling Flows. In recent years, multiple works have been proposed in the graph representation learning field that integrate controlling into neural networks to achieve flexible data-driven control. For example, DGCNN (Zhang et al., 2018) implemented a differentiable sort operator (sort pooling) to build more powerful readout functions. Graph U-Net (Gao & Ji, 2019; Cangea et al., 2018) designed an adaptive pooling operator (TopK pooling) to support flexible data-driven pooling operations. All these methods achieved the differentiability by relaxing and multiplying the controlling signals with the neural networks' hidden representations. Inspired by their works, our method also differentiates the iterative algorithm by relaxing and multiplying the stopping criterion's output into neural networks' hidden representations.

Adaptive Depth of Neural Networks. The final formulation of our method is generally similar to the previous adaptive computation time algorithm (ACT) (Graves, 2016) for RNNs or spatially ACT (Figurnov et al., 2017; Eyzaguirre & Soto, 2020) for CNNs, however, with distinct motivations and formulation details. The numbers of iterations for ACT are usually small by design (e.g.the formulation of regularizations and halting distributions). Other than to achieve more representation powers through repeating operations for multiple times, or to achieve more efficiency by selectively iterating for fewer times, our method is designed to fundamentally improve the generalizability of GNNs w.r.t. scales by generalizing to much larger iteration numbers. Several improvements are proposed accordingly. Recently, there are also flow-based methods that (e.g. the Graph Neural ODE (Poli et al., 2019)) are also potentially able to provide adaptive layer numbers. However, with no explicit iteration controller, they are not a straightforward solution to approximate iterative algorithms and to encode related inductive biases. For example, it is not easy to transform the solution of the shortest path into an ODE.

F. Backgrounds - Graph Neural Networks

In this section, we briefly describe the graph neural networks (GNNs). We first present the GN blocks, which generalize many GNN layers, in Section F.1 and then present the common practice of building GNN models for graph classification/regression in Section F.2. The notations and terms are further utilized while describing PathGNN layers in Section B.2 and while describing the models/baselines in our experiments in Section C.4.

F.1. Graph Network Blocks (GN blocks)

We briefly describe a popular framework to build GNN layers, called Graph Network blocks (GN blocks) (Battaglia et al., 2018). It encompasses our PathGNN layers presented in Section B.2 as well as many state-of-art GNN layers, such as GCN (Kipf & Welling, 2017), GAT (Veličković et al., 2018), and GIN (Xu et al., 2019). Readers are referred to (Battaglia et al., 2018) for more details. Note that we adopt different notations from the main body to be consistent with (Battaglia et al., 2018). Also note that, even when the global attribute is utilized, the fixed-depth fixed-width GNNs still lose a significant portion of powers for solving many graph problems as proved in (Loukas, 2020). For example, the minimum depth of GNNs scales sub-linearly with the graph sizes for accurately verifying a set of edges as the shortest path or a s-t cut, given the message passing nature of GNNs.

The input graphs are defined as $G = (\mathbf{u}, V, E)$ with node attributes $V = \{\mathbf{v}_i, i = 1, 2, \ldots, N_v\}$, edge attributes $E = \{(\mathbf{e}_k, s_k, r_k), k = 1, 2, \ldots, N_e\}$, and the global attribute $\mathbf{u}$, where s_k and r_k denote the index of the sender and receiver nodes for edge k, $\mathbf{u}$, $\mathbf{v}_i$, and $\mathbf{e}_k$ represent the global attribute vector, the attribute vector of node i, and the attribute vector of edge k, respectively.

The GN block performs message passing using three update modules ϕ^e, ϕ^v, ϕ^u and three aggregation modules $\rho^{e\to v}, \rho^{e\to u}, \rho^{v\to u}$ as follows:

1. Node s_k sends messages $\mathbf{e}'_k$ to the receiver node r_k, which are updated according to the related node attributes $\mathbf{v}$, edge attributes $\mathbf{e}$, and global attributes $\mathbf{u}$.

$$\mathbf{e}'_k = \phi^e(\mathbf{e}_k, \mathbf{v}_{r_k}, \mathbf{v}_{s_k}, \mathbf{u}), k = 1, 2, \ldots, N_e$$

2. For each receiver node i, the sent messages are aggregated using the aggregation module.

$$\bar{\mathbf{e}}'_i = \rho^{e\to v}(\{\mathbf{e}'_k | r_k = i\}$$

3. The aggregated messages are then utilized for updating the node attribute together with the related node attributes $\mathbf{v}_i$ and global attributes $\mathbf{u}$.

$$\mathbf{v}'_i = \phi^v(\bar{\mathbf{e}}'_i, \mathbf{v}_i, \mathbf{u})$$

4. The sent messages $\mathbf{e}'_k$ can also be aggregated for updating the global attribute $\mathbf{u}$ as

$$\bar{\mathbf{e}}' = \rho^{e\to u}(\{\mathbf{e}'_k, k = 1, 2, \ldots, N_e\}$$

5. The node attribute $\mathbf{v}'_i$ can be aggregated for updating the global attribute $\mathbf{u}$ as well.

$$\bar{\mathbf{v}}' = \rho^{v\to u}(\{\mathbf{v}'_i, i = 1, 2, \ldots, N_v\}$$

6. The global attribute $\mathbf{u}$ are updated according to the aggregated messages $\bar{\mathbf{e}}'$, aggregated node attributes $\bar{\mathbf{v}}'$, and previous global attribute $\mathbf{u}$.

$$\mathbf{u}' = \phi^u(\bar{\mathbf{e}}', \bar{\mathbf{v}}', \mathbf{u})$$

ϕ^e is often referred as the message module, $\rho^{e\to v}$ as the aggregation module, ϕ^v as the update module, $\rho^{e\to u}$ and $\rho^{v\to u}$ as the readout modules, in many papers. The three update modules are simple vector-to-vector modules such as multi-layer perceptrons (MLPs). The three aggregation modules, on the other hand, should be permutation-invariant functions on sets such as max pooling and attentional pooling (Li et al., 2016; Lee et al., 2019).

F.2. Composing GN blocks for graph classification / regression

We follow the common practice (Battaglia et al., 2018; Zhang et al., 2018; Xu et al., 2019) in the field of supervised graph classification while building models and baselines in our experiments. Typically, the models are built by sequentially stacking the node-wise embedding module, the core GNN module, the readout function, and the task-specific prediction module. The node-wise embedding module corresponds to GN blocks that are only composed of function ϕ^v for updating node attributes. More intuitively, it applies the same MLP module to update all node attribute vectors. The core GNN module performs message passing to update all attributes of graphs. The readout function corresponds to GN blocks that only consist of the readout modules such as $\rho^{e\to u}$ and $\rho^{v\to u}$ to summarize information of the whole graph into a fixed-dimensional vector $\mathbf{u}$. The task-specific prediction module then utilizes the global attribute vector $\mathbf{u}$ to perform the final prediction, such as predicting the number of connected components or the Q-values within the symbolic Pacman environment.

We need to specify three properties while designing the core GNN modules: (1) the internal structure of GN blocks; (2) the composition of GN blocks; and (3) the prior that encodes the properties of the solutions of the problem.

- The internal structure of GN blocks defines the logic about how to perform one step of message passing. It is usually specified by selecting or designing the GNN layers.
- The composition of GN blocks defines the computational flow among GN-blocks. For example, the simplest paradigm is to apply multiple GN blocks sequentially. Our iterative module introduces the iterative architecture into GNNs. It applies the same GN block for multiple times. The iteration number is adaptively determined by our iterative module.
- The prior is usually specified by adopting the regularization terms. For example, regularizing the L2 norm of weights can encode the prior that GNNs representing solutions of the problem usually have weights of small magnitudes. Regularizing the L1 norm of weights can encode prior about sparsity. We can utilize the HomoMLP and HomoGNN as described in the main body to encode the homogeneous prior that the solutions of most classical graph problems are homogeneous functions.