nature machine intelligence

Supplementary information

https://doi.org/10.1038/s42256-024-00833-7

Distributed constrained combinatorial optimization leveraging hypergraph neural networks

In the format provided by the authors and unedited

Supplementary Information

CONTENTS

I. Related Work

The recent significant progress in AI and ML has prompted their applications in optimization [1], [2], [3], [4]. Combinatorial optimization, given its intricate nature, stands to benefit greatly from learning-based approaches [1], [5], [6], [7], [8]. Learning methods for combinatorial optimization problems can be categorized into supervised learning [9], [10], [11], [12], [13], unsupervised learning [8], [14], [15], and reinforcement learning [7], [16], [17], [18]. Although supervised learning methods have shown promising results in various combinatorial optimization problems, they need a dataset of solved instances, posing challenges of computational intensity, expense, and limited generalizability to new or larger problems. Reinforcement learning is a powerful tool for optimization and has been proved to be effective for many problems. However, training RL models is usually challenging and time-consuming. Therefore, RL methods are best to be used for finding efficient algorithms to solve specific problems [19]. Unsupervised learning methods enable fast and generalizable optimization of combinatorial optimization problems without requiring a dataset of solved problem instances. One issue with unsupervised learning methods is that they are based on gradient descent and therefore, they may converge to subpar local optima.

To address the issues of the unsupervised learning methods getting stuck at local optima, [20] proposed an annealed training framework for combinatorial optimization problems over graphs. A smoothing term is added to the loss function to help the training process escape the local optima. This smoothing strategy can be combined with any unsupervised learning method for combinatorial optimization including the method described in this paper.

While there is no existing research that models constrained combinatorial optimization problems as hypergraphs to capture higher-level dependencies between variables, a study has approached maximum constraint satisfiability problems by modeling them as bipartite graphs and training a Graph Neural Network (GNN) to find solutions [15]. However, it is limited to problems with constraints involving only two variables and is not suitable for problems with different types of constraints, such as 3-SAT.

A. Distributed Training

Numerous research works have addressed the acceleration and scalability of Graph Neural Network (GNN) training in distributed and parallel computing. Broadly categorized, these efforts focus on system-level optimizations, modifying graph data preprocessing and execution processes, or algorithm-level enhancements proposing efficient machine learning algorithms with minimal inter-server communication and improved model aggregation strategies [21]. The system-level approaches are orthogonal to HypOp and can be incorporated within HypOp to further accelerate the training process. At the algorithm-level [21], works like [22], [23], [24], [25] have focused on optimizing graph partitioning or model aggregation strategies for accelerated GNN training. HypOp doesn't enhance training by optimizing the graph partitioning scheme since it is designed for scenarios where the graph is inherently distributed across multiple servers and cases where the graph is partitioned beforehand. In this context, graph partitioning schemes are orthogonal to HypOp and can be incorporated for improved scalability in the latter scenario. Furthermore, HypOp exhibits two advantages over these approaches: (i) Unlike partition-based methods [22], [23], [24], HypOp doesn't require subgraph sampling at every iteration, thereby reducing significant communication overhead; (ii) [23] proposes a superior aggregation strategy to reduce communication overhead, however, it requires longer iterations for convergence. In contrast, HypOp achieves comparable results within the same epochs as single GPU training, showcasing its efficiency and scalability for parallel/distributed training.

II. Preliminaries: Hypergraph Neural Networks

A hypergraph is a generalized version of a graph in which, instead of having edges connecting two nodes, we have hyperedges connecting multiple nodes to each other, allowing for more complex relationships than traditional graphs. Hypergraph neural network (HyperGNN) [26] is an advanced variation of graph neural networks that is designed to handle hypergraphs. HyperGNNs extend the concept of GNNs to hypergraphs by introducing new aggregation mechanisms that consider higher-order connections (hyperedges). HyperGNNs can capture intricate dependencies among multiple nodes simultaneously, making them well-suited for tasks involving data with complex associations, such as social interactions, collaboration data, knowledge graphs, and recommendation systems. In hypergraph convolutional networks introduced in [26], features of nodes are first aggregated in each hyperedge and are then passed on to new nodes for another aggregation. In particular, we have the following layer-wise operation:

$$H^{(l+1)} = \sigma(D_v^{-\frac{1}{2}} A D_e^{-1} A^T D_v^{-\frac{1}{2}} H^{(l)} W^{(l)}) \tag{1}$$

Algorithm 2 Parallel Multi-GPU HyperGNN Training

Require: Hypergraph $G = (V, E)$, HyperGNN Model M, Number of GPUs S, Epoch Number EP
Ensure: Trained HyperGNN Model M

```
Initialize M on each GPU
for ep in EP do
    Shuffle E to obtain E_shuffled
    Partition E_shuffled into S sublists: E^1_shuffled, E^2_shuffled, ..., E^S_shuffled
    for s ← 1 to S do
        Assign E^s_shuffled to GPU^s
        Train M^s on GPU^s with edges E^s_shuffled
    end for
    Aggregate gradients from all GPUs and update M
    Broadcast updated M to all GPUs
end for
return M
```

where $H^{(l)} \in \mathbb{R}^{N \times D}$ is the matrix of node features at l_{th} layer, A is the hypergraph incidence matrix defined as $(A)_{ij} = 1$ if node i belongs to hyperedge j, and $(A)_{ij} = 0$ otherwise. D_v and D_e are the diagonal degree matrices of nodes and hyperedges, respectively, and $W^{(l)}$ is the l_{th} layer trainable weight matrix.

III. HyperGCN Structure

In our HyperGCN model, we use two convolutional layers with a Relu activation function after the first layer. For the case of binary variables, we use Sigmoid activation function to generate a number between 0 and 1. The dimension of the convolutional layers are chosen as follows. The first layer has an f dimensional input and outputs an $f/2$ dimensional vector. The second layer's input and output dimensions are $f/2$ and 1, respectively. For most of out experiments, we set $f = \sqrt{N}$, where N is the number of variables.

IV. Black-Box Optimization of the Hyperparameters

We have incorporated the optional hyperparameter optimization step in HypOp, in which we conduct a sampling based black-box optimization w.r.t. the hyperparameters of the algorithm to improve the performance of HypOp. We use AdaNS optimization tool [27] for this step. One of the hyperparameters that we optimize over is the initial value of the input embedding of the HyperGNN. Since our method is based on gradient descent, the initialization point can potentially be important in avoiding bad local optima and therefore, the output of the algorithm can be improved if we optimize w.r.t. the initial point. Other potential hyperparameters that we can optimize with this tool are the learning rate and the SA hyperparameters.

V. Real and Synthetic Hypergraphs

The real hypergraphs used in our experiments are extracted from the the American Physical Society (APS) [28]. The APS dataset contains the basic metadata of all APS journal articles from year 1993 to 2021. Focusing on a specific journal within this dataset, Physical Review E (PRE), we partitioned the data into 3-year segments, and extracted the giant connected component of the authorship hypergraphs. The synthetic hypergraphs are generated randomly for specific values of number of nodes, edges, and upper bounds on the node degrees.

VI. National Drug Code (NDC) Dataset

NDC dataset consists of the substances that make up drugs in the National Drug Code Directory maintained by the Food and Drug Administration [29], [30]. This dataset consists of a list of drugs and the substances that make them. The drugs are considered as nodes and substances are presented by hyperedges. Drugs (nodes) within each hyperedge contain that substance. We considered the experiment of choosing a set of substances such that if one puts some regulatory measures on them, the rest of the dataset is divided into two separate (smaller) isolated

group of drugs that can be addressed separately. We are interested to know the maximum number of substances that we need to regulate to create two separate datasets. This experiment can be represented by a hypergraph MaxCut problem. Before applying HypOp on NDC dataset, We first removed the hyperedges containing one node and then removed the isolated nodes. As a result, we constructed a hypergraph with 3767 nodes and 29810 hyperedges.

VII. Experiment Results

A. Graph MaxCut

We have considered graph MaxCut problem to compare HypOp with the baseline solver [8] (PI-GNN). Similar to PI-GNN, we run HypOp on Gset dataset [31]. Table I summarizes our experiments on this dataset. As can be seen in the table, HypOp outperforms PI-GNN in all of the tested graphs.

We need to note here that Graph MaxCut is a very well studied problem and efficient heuristic algorithms have been developed to solve this problem. Although one can use algorithms such as HypOp and PI-GNN to solve MaxCut problems, these algorithms are not expected to outperform the efficient heuristics that are customized to the MaxCut problem specifics. However, the advantage of the learning-based algorithms such as HypOp and PI-GNN is that they are general solvers and one can use the same tool to solve thousands of optimization problems without the need to learn to run specific heuristics for each problem. So the main point of benchmarking HypOp on MaxCut problem is to compare its performance with the baseline algorithm PI-GNN and to further motivate the additional components that we have added to the baseline method.

Graph	G14	G15	G22	G49	G50	G55
Nodes	800	800	2000	3000	3000	5000
Edges	4694	4661	19990	6000	6000	12498
Run-CSP	2943	2928	13028	6000	5880	10116
PI-GNN	3026	2990	13181	5918	5820	10,138
HypOp	3042	3030	13185	5999	5879	10150
Best Known Result	3064	3050	13359	6000	5880	10,294

Supplementary Table I: Graph MaxCut with HypOp. Performance of HypOp for graph MaxCut problem on the Gset dataset. HypOp outperforms other unsupervised learning based methods, while achieving results that are close to the best known heuristics.

B. SAT Problems

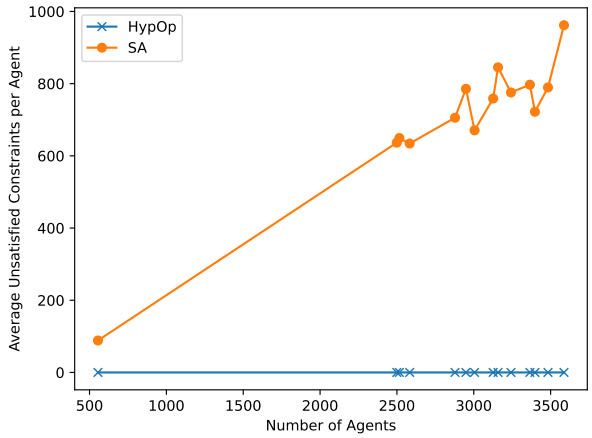
We have tested our algorithm on random 3-SAT benchmarks from SATLIB dataset [32]. Table II in the supplementary section shows the performance of HypOp on some of the satisfiable instances. We see that except for one with less than 2% violation of constraints, all others were successfully solved. We provide our experiments on SAT problems to showcase the success of our algorithm on various problem instances, including the challenging and widely studied SAT problems. We need to note that we do not expect HypOp to compete with the powerful heuristics that are designed specifically for SAT problems and therefore, we do not provide any baselines in this section.

SAT Dataset	uf 20-91	uf 100-430	uf 200-860	uf 250-1065
Number of problems	100	100	100	100
Percentage Solved	100%	100%	100%	98%
Average number of unsatisfied clauses	0	0	0	1
Average time	4s	15s	37s	60s
Median time	4s	14s	21s	29s

Supplementary Table II: SAT with HypOp. Performance of HypOp for random 3-SAT problems in the SATLIB dataset.

C. Resource Allocation Problem

We use HypOp to solve a resource allocation problem where there are a number of tasks to be assigned to a number of agents with given energy and budget constraints. In particular, each agent has an energy budget to spend on the tasks and each task requires a minimum energy to be completed. We can also consider a global arbitrary objective function to minimize while satisfying the constraints. One can model this problem as a hypergraph discovery problem where the hypergraph nodes correspond to the agents and the hyperedges correspond to the tasks. The nodes (agents) within each hyperedge (task) are assigned to that task. One needs to discover different variations of this hypergraph to find feasible solutions and optimize the objective function. For simplicity, we do not consider a global objective function in our experiments and only try to find feasible solutions. In order to get realistic energy and budget constraints, we consider the APS co-authorship hypergraphs [28]. The papers correspond to tasks and the authors correspond to agents. The energy requirement of each task (paper) is the number of authors in it. Each agent has a budget that is a fixed number more than the papers that the agent appears in. We refer to this number as the budget surplus. The feasible set would be larger if we increase the budget surplus. We use the same budget surplus for all of the hypergraphs that we have considered so that as the number of agents in the hypergraph increases, the size of the feasible set shrinks. We use the initial co-authorship hypergraphs to build the HyperGNN in HypOp and try to find feasible solutions by assigning agents to tasks. We compare the result of HypOp with SA in Supplementary Figure 1. As can be seen in the figure, SA fails to find feasible solutions and its performance gets worse as the size of the feasible set shrinks (the number of agents increases). However, HypOp can find feasible solutions for all number of agents. Furthermore, in terms of run time, HypOp can solve these resource allocation problems in a significantly less time than SA (See Table III).



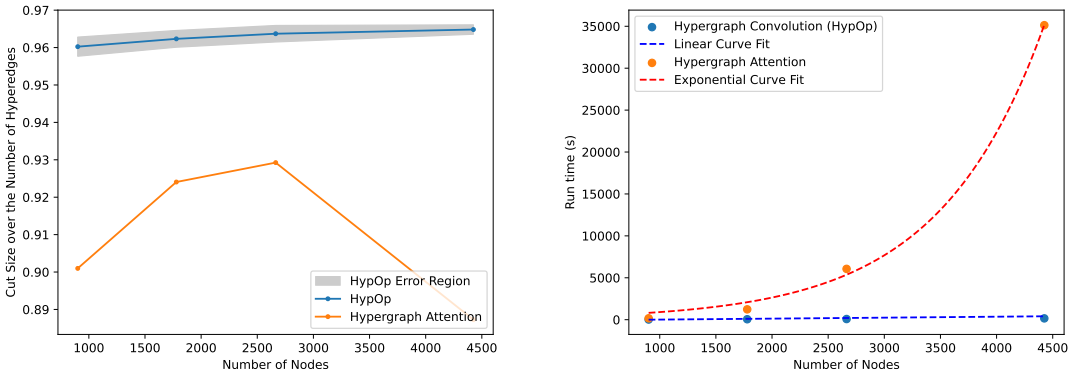
Supplementary Figure 1: Resource Allocation with HypOp. Unsatisfied constraints in the Resource Allocation problem with different number of agents. HypOp can find feasible solutions (zero constraint violation), while SA fails to do so.

Supplementary Table III: Average run time of HypOp for resource allocation problems.

	Run Time (s)
HypOp	4668 ± 2017
SA	35472 ± 61448

VIII. HyperGNN Architecture Choice

Two prominent HyperGNN architectures are Hypergraph Convolutional Networks (HyperGCNs) and Hypergraph Attention Networks (HyperGATs). HyperGCNs extend traditional graph convolutional networks to hypergraphs, enabling the incorporation of higher-order relationships. On the other hand, HyperGATs leverage attention mechanisms to weigh the importance of different hyperedges and nodes, allowing the network to focus on relevant information during message passing. In HypOp, we opted for HyperGCN architecture due to the unsupervised nature of our approach and the fact that they facilitate easier training for each problem instance. To support our choice, we also implemented and tested a HyperGAT architecture within HypOp. The HyperGAT model is based on the implementation in [33]. In Supplementary Figure 2, we show the performance and runtime of HypOp with



(a) Performance **(b)** Run time

Supplementary Figure 2: HyperGCN vs. HyperGAT. Performance of HypOp with HyperGCN architecture, compared with HyperGAT for hypergraph MaxCut problem on synthetic random hypergraphs. HyperGAT can not achieve the same performance compared to HyperGCN, while requiring significantly more time to train. HypOp performance is presented as the average of the results from 10 sets of experiments and the error region shows the standard deviation of the results.

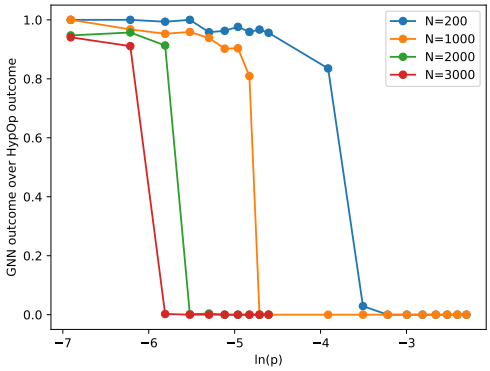
HyperGCN against the same method with a HyperGAT architecture. As depicted in the figure, HyperGAT can not achieve the same performance compared to HyperGCN, while requiring significantly more time to train.

IX. Limitations: Phase Transition in GNN Performance

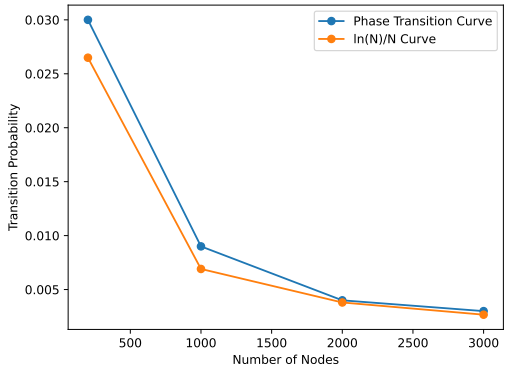
Although learning-based approaches for optimization have shown tremendous potential, one has to be mindful of their limitations. In this section, we provide our experimental results on the performance of GNNs for MIS problems on various graphs. Note that for the analysis in this section, we consider the GNN outcome in HypOp with a simple threshold mapping as used in PI-GNN. For problems on graphs (including MIS), this will be the same as the PI-GNN result. We compare this result to the fine-tuned outcome generated by HypOp as an end-to-end solver. Our goal is to show the limitations of PI-GNN on some graphs. This analysis is inspired by the results of [34], where the authors study the limitations of the network online learning algorithms by considering different types of graphs.

We investigated different structures of graphs, namely, Erdos-Renyi Random graphs, Regular graphs, and Power-law graphs (graphs with powerlaw degree distribution), with various options for their number of nodes and edges. We observed that for MIS problem, PI-GNN can not learn anything on the graphs that are denser than a given threshold. In Supplementary Figure 3(a) and 3(c), we show the performance of PI-GNN on Erdos-Renyi random graphs for different parameters, p, and different number of nodes, N, for two different learning rates. As is evident in the figure, the performance drops to 0 at a given threshold. The threshold, however, varies with the number of nodes. In Supplementary Figure 3(b) and 3(d), we show the plot of the phase transition threshold w.r.t. the number of nodes. The Erdos-Renyi graphs are well known for their phase transition in their connectivity. The threshold for this phase transition is $p^* = \frac{\ln(N)}{N}$. As we have shown in Supplementary Figure 3(b) and 3(d), the phase transition threshold in the performance of PI-GNN is almost the same as the threshold for connectivity phase transition. This means that PI-GNN does not learn a solution over graphs that are dense enough to be connected. We provide our experimental results for two different learning rates to show that the performance drop is not caused by a high learning rate and is instead caused by the potential inability of GNNs to learn on dense graphs.

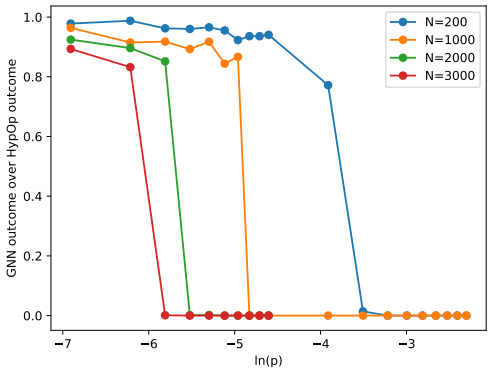
In order to investigate whether or not the specific structure of the graphs plays a role in the observed phase transition, we have compared the performance of PI-GNN on Powerlaw graphs, Regular graphs, and Random (Erdos-Renyi) graphs. In order to do the comparison, for each parameter of the powerlaw graphs, we construct random and regular graphs with almost the same number of edge density and compare the performance of PI-GNN



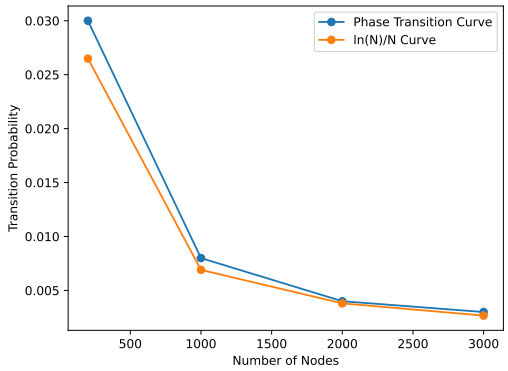
(a) Phase transition in the performance of PI-GNN over random graphs.



(b) Phase transition threshold w.r.t. the graph size, and comparison with the Erdos-Renyi graph connectivity phase transition, $p^* = \frac{ln(N)}{N}$.



(c) Phase transition in the performance of PI-GNN over random graphs.

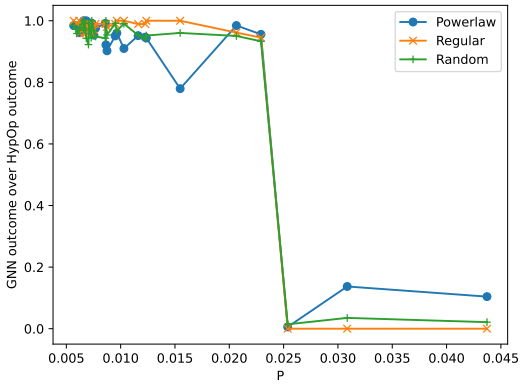


(d) Phase transition threshold w.r.t. the graph size, and comparison with the Erdos-Renyi graph connectivity phase transition, $p^* = \frac{ln(N)}{N}$.

Supplementary Figure 3: Phase Transition in GNN Learning. Phase transition in the GNN-based optimization performance for MIS problem with learning rate of $1e-4$ in (a) and (b) and learning rate of $1e-5$ in (c) and (d).

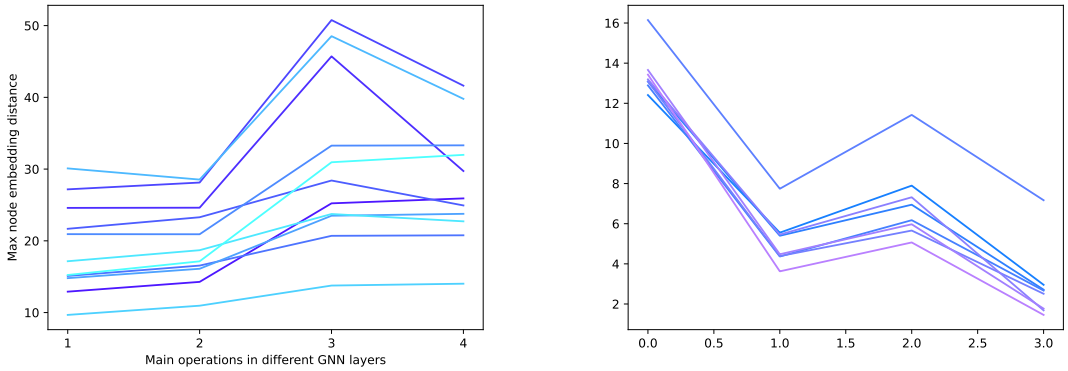
over the three generated graphs. In Supplementary Figure 4, we show that the phase transition is the same for all of the three types of graphs when the performance is plotted w.r.t. the density (p) of the Random graph. Therefore, based on the existing evidence, we conclude that the performance is affected mainly by the density of the graph as opposed to its structure. We need to note here that most of the real graphs are powerlaw graphs with components between 2 to 4. Such graphs usually have lower densities than the observed critical density threshold for GNN learning.

The phase transition in the GNN learning can be attributed to the oversmoothing issue that is widely observed in the GNNs throughout the literature [35]. Roughly speaking, oversmoothing occurs due to too many feature averaging, leading all hidden features to converge to the same point. In Supplementary Figure 5, we show the maximum distance between the output embeddings of the different nodes after the main operations of the HyperGNN (two convolutional and two aggregation operators). We provide two saparate figures, one for the graphs for which the PI-GNN is successful in producing good outcomes (graphs that have density less than the threshold found in Supplementary Figure 3), and one for the graphs for which the PI-GNN is not successful in producing good outcomes (graphs that are denser than the threshold found in Supplementary Figure 3). We see that the embedding distance



Supplementary Figure 4: Phase Transition in GNN Learning. Phase transition in the performance of PI-GNN for Powerlaw, Regular, and Erdos-Renyi graphs.

is decreasing and converges to a small number for the dense graphs. This observation confirms the occurrence of oversmoothing in the HyperGNN training for dense graphs.



(a) Successful GNN training: graphs that are sparser than the phase transition threshold.

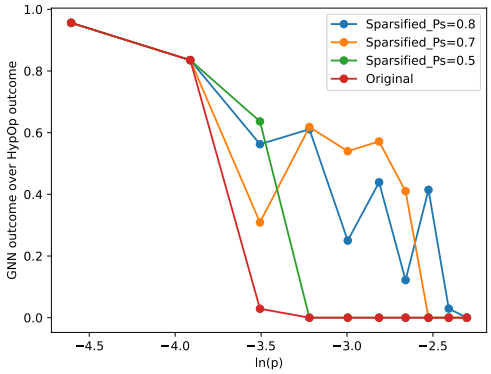
(b) Unsuccessful GNN training: graphs that are denser than the phase transition threshold.

Supplementary Figure 5: Oversmoothing. Maximum node embedding distance after four main GNN operations. The plot is generated for MIS problem over random graphs with $N = 200$. Oversmoothing is happening for graphs in (b) where the GNN training is unsuccessful.

A solution proposed in the literature for oversmoothing is graph sparsification by dropping edges [36]. This also explains how GNNs fail to learn good solutions on dense graphs as observed in Supplementary Figure 3. Motivated by [36], we propose graph sparsification to improve the GNN's performance in HypOp.

A. Graph Sparsification

With the observation that the inability of the GNNs to learn a good solution on some graphs is attributed to their high density, we explored the possibility of improving GNN learning through the process of graph sparsification. We experimented with the same graphs of Supplementary Figure 3 with $N = 200$ nodes and sparsified the ones that the GNN failed to learn a good solution by removing each edge with a given probability Ps. A similar method was conducted in [36] to alleviate the oversmoothing issue in graph neural networks. We show the performance of



Supplementary Figure 6: Graph Sparsification. The performance of PI-GNN over original and sparsified random graphs with $N = 200$ for MIS problem.

HypOp on the sparsified graphs with different values of Ps in Supplementary Figure 6. We can see in the figure that sparsification improves the performance of PI-GNN. However, there is still a gap between the overall outcome quality of HypOp and the outcome of the PI-GNN.

Based on the findings presented in this section, we conclude that integrating Graph Neural Networks (GNNs) and HyperGNNs with other solvers yields significant advantages compared to employing them as standalone end-to-end solvers. This observation serves as additional motivation for the inclusion of the fine-tuning step using Simulated Annealing (SA) in HypOp. This supplementary fine-tuning step ensures a reliable solution, even in cases where the GNN fails to learn a good solution.

X. Effect of Self-Loops in the HyperGCN

It is common to add self-loops to the graphs before training a GNN model on them. We observed that in HypOp (and similarly in PI-GNN), adding self loops will deteriorate the performance of the GNN or HyperGNN. The reason is most likely related to the results shown in the previous section where we showed that the GNNs fail to learn on graphs that are denser than a given threshold. Adding self loops will make the graphs denser than is tolerated by the GNN, and therefore, we did not add self loops and also set the diagonal terms in the hypergraph adjacency matrix to 0 to create sparser graphs and hypergraphs.

REFERENCES

[1] Yoshua Bengio, Andrea Lodi, and Antoine Prouvost. Machine learning for combinatorial optimization: a methodological tour d'horizon. *European Journal of Operational Research*, 290(2):405–421, 2021.

[2] Chris Cummins, Pavlos Petoumenos, Zheng Wang, and Hugh Leather. End-to-end deep learning of optimization heuristics. In *2017 26th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 219–232. IEEE, 2017.

[3] Lu Liu, Yu Cheng, Lin Cai, Sheng Zhou, and Zhisheng Niu. Deep learning based optimization in wireless network. In *2017 IEEE international conference on communications (ICC)*, pages 1–6. IEEE, 2017.

[4] Ying He, Zheng Zhang, F Richard Yu, Nan Zhao, Hongxi Yin, Victor CM Leung, and Yanhua Zhang. Deep-reinforcement-learning-based optimization for cache-enabled opportunistic interference alignment wireless networks. *IEEE Transactions on Vehicular Technology*, 66(11):10433–10445, 2017.

[5] Bingjie Li, Guohua Wu, Yongming He, Mingfeng Fan, and Witold Pedrycz. An overview and experimental study of learning-based optimization algorithms for the vehicle routing problem. *IEEE/CAA Journal of Automatica Sinica*, 9(7):1115–1138, 2022.

[6] Quentin Cappart, Didier Chételat, Elias B Khalil, Andrea Lodi, Christopher Morris, and Petar Veličković. Combinatorial optimization and reasoning with graph neural networks. *Journal of Machine Learning Research*, 24(130):1–61, 2023.

[7] Azalia Mirhoseini, Anna Goldie, Mustafa Yazgan, Joe Wenjie Jiang, Ebrahim Songhori, Shen Wang, Young-Joon Lee, Eric Johnson, Omkar Pathak, Azade Nazi, et al. A graph placement methodology for fast chip design. *Nature*, 594(7862):207–212, 2021.

[8] Martin JA Schuetz, J Kyle Brubaker, and Helmut G Katzgraber. Combinatorial optimization with physics-inspired graph neural networks. *Nature Machine Intelligence*, 4(4):367–377, 2022.

[9] Elias Khalil, Pierre Le Bodic, Le Song, George Nemhauser, and Bistra Dilkina. Learning to branch in mixed integer programming. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 30, 2016.

[10] Yunsheng Bai, Hao Ding, Song Bian, Ting Chen, Yizhou Sun, and Wei Wang. Simgnn: A neural network approach to fast graph similarity computation. In *Proceedings of the twelfth ACM international conference on web search and data mining*, pages 384–392, 2019.

[11] Maxime Gasse, Didier Chételat, Nicola Ferroni, Laurent Charlin, and Andrea Lodi. Exact combinatorial optimization with graph convolutional neural networks. *Advances in neural information processing systems*, 32:15580–15592, 2019.

[12] Vinod Nair, Sergey Bartunov, Felix Gimeno, Ingrid Von Glehn, Pawel Lichocki, Ivan Lobov, Brendan O'Donoghue, Nicolas Sonnerat, Christian Tjandraatmadja, Pengming Wang, et al. Solving mixed integer programs using neural networks. *arXiv preprint arXiv:2012.13349*, 2020.

[13] Zhuwen Li, Qifeng Chen, and Vladlen Koltun. Combinatorial optimization with graph convolutional networks and guided tree search. *Advances in neural information processing systems*, 31:537–546, 2018.

[14] Nikolaos Karalias and Andreas Loukas. Erdos goes neural: an unsupervised learning framework for combinatorial optimization on graphs. *Advances in Neural Information Processing Systems*, 33:6659–6672, 2020.

[15] Jan Toenshoff, Martin Ritzert, Hinrikus Wolf, and Martin Grohe. Graph neural networks for maximum constraint satisfaction. *Frontiers in artificial intelligence*, 3:580607, 2021.

[16] Emre Yolcu and Barnabás Póczos. Learning local search heuristics for boolean satisfiability. *Advances in Neural Information Processing Systems*, 32:7992–8003, 2019.

[17] Qiang Ma, Suwen Ge, Danyang He, Darshan Thaker, and Iddo Drori. Combinatorial optimization by graph pointer networks and hierarchical reinforcement learning. *arXiv preprint arXiv:1911.04936*, 2019.

[18] Wouter Kool, Herke van Hoof, and Max Welling. Attention, learn to solve routing problems! In *International Conference on Learning Representations*, 2018.

[19] Alhussein Fawzi, Matej Balog, Aja Huang, Thomas Hubert, Bernardino Romera-Paredes, Mohammadamin Barekatain, Alexander Novikov, Francisco J R Ruiz, Julian Schrittwieser, Grzegorz Swirszcz, et al. Discovering faster matrix multiplication algorithms with reinforcement learning. *Nature*, 610(7930):47–53, 2022.

[20] Haoran Sun, Etash Kumar Guha, and Hanjun Dai. Annealed training for combinatorial optimization on graphs. In *OPT 2022: Optimization for Machine Learning (NeurIPS 2022 Workshop)*, 2022.

[21] Haiyang Lin, Mingyu Yan, Xiaochun Ye, Dongrui Fan, Shirui Pan, Wenguang Chen, and Yuan Xie. A comprehensive survey on distributed training of graph neural networks. *Proceedings of the IEEE*, 2023.

[22] Wei-Lin Chiang, Xuanqing Liu, Si Si, Yang Li, Samy Bengio, and Cho-Jui Hsieh. Cluster-gcn: An efficient algorithm for training deep and large graph convolutional networks. In *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining*, pages 257–266, 2019.

[23] Jiyang Bai, Yuxiang Ren, and Jiawei Zhang. Ripple walk training: A subgraph-based training framework for large and deep graph neural network. In *2021 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8. IEEE, 2021.

[24] Hanqing Zeng, Hongkuan Zhou, Ajitesh Srivastava, Rajgopal Kannan, and Viktor Prasanna. Graphsaint: Graph sampling based inductive learning method. In *International Conference on Learning Representations*, 2019.

[25] Morteza Ramezani and Weilin Cong. Learn locally, correct globally: A distributed algorithm for training graph neural networks. In *International Conference on Learning (ICLR)*, 2022.

[26] Yifan Feng, Haoxuan You, Zizhao Zhang, Rongrong Ji, and Yue Gao. Hypergraph neural networks. In *Proceedings of the AAAI conference on artificial intelligence*, volume 33, pages 3558–3565, 2019.

[27] Mojan Javaheripi, Mohammad Samragh, Tara Javidi, and Farinaz Koushanfar. Adans: Adaptive non-uniform sampling for automated design of compact dnns. *IEEE Journal of Selected Topics in Signal Processing*, 14(4):750–764, 2020.

[28] Aps dataset on Physical Review journals. Published by the American Physical Society, https://journals.aps.org/datasets.

[29] Ndc-substances dataset. https://www.cs.cornell.edu/ arb/data/NDC-substances/.

[30] Austin R. Benson, Rediet Abebe, Michael T. Schaub, Ali Jadbabaie, and Jon Kleinberg. Simplicial closure and higher-order link prediction. *Proceedings of the National Academy of Sciences*, 2018.

[31] Y. Ye. The gset dataset. https://web.stanford.edu/ yyye/ yyye/Gset/ (Stanford, 2003).

[32] Holger H Hoos and Thomas Stützle. Satlib: An online resource for research on sat. *Sat*, 2000:283–292, 2000.

[33] Kaize Ding, Jianling Wang, Jundong Li, Dingcheng Li, and Huan Liu. Be more with less: Hypergraph attention networks for inductive text classification. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 4927–4936, 2020.

[34] Timothy LaRock, Timothy Sakharov, Sahely Bhadra, and Tina Eliassi-Rad. Understanding the limitations of network online learning. *Applied Network Science*, 5:1–25, 2020.

[35] T Konstantin Rusch, Michael M Bronstein, and Siddhartha Mishra. A survey on oversmoothing in graph neural networks. *arXiv preprint arXiv:2303.10993*, 2023.

[36] Yu Rong, Wenbing Huang, Tingyang Xu, and Junzhou Huang. Dropedge: Towards deep graph convolutional networks on node classification. In *International Conference on Learning Representations*, 2019.