Simplifying Architecture Search for Graph Neural Network

Huan Zhao[a], Lanning Wei[b] and Quanming Yao[c]

[a] 4Paradigm Inc. Shenzhen
[b] Institute of Computing Technology Chinese Academy of Sciences
[c] Hong Kong

Abstract

Recent years have witnessed the popularity of Graph Neural Networks (GNN) in various scenarios. To obtain optimal data-specific GNN architectures, researchers turn to neural architecture search (NAS) methods, which have made impressive progress in discovering effective architectures in convolutional neural networks. Two preliminary works, GraphNAS and Auto-GNN, have made first attempt to apply NAS methods to GNN. Despite the promising results, there are several drawbacks in expressive capability and search efficiency of GraphNAS and Auto-GNN due to the designed search space. To overcome these drawbacks, we propose the SNAG framework (Simplified Neural Architecture search for Graph neural networks), consisting of a novel search space and a reinforcement learning based search algorithm. Extensive experiments on real-world datasets demonstrate the effectiveness of the SNAG framework compared to human-designed GNNs and NAS methods, including GraphNAS and Auto-GNN.[1]

1. Introduction

In recent years, Graph Neural Networks (GNN) [1, 2] have been a hot topic due to their promising results on various graph-based tasks, e.g., recommendation [3, 4, 5], fraud detection [6], chemistry [7]. In the literature, various GNN models [8, 9, 10, 11, 12, 13, 6, 5] have been designed for graph-based tasks. Despite the success of these GNN models, there are two challenges facing them. The first one is that there is no optimal architecture which can always behave well in different scenarios. For example, in our experiments (Table 3 and 4), we can see that the best GNN architectures vary on different datasets and tasks. It means that we have to spend huge computational and expertise resources designing and tuning a well-behaved GNN architecture given a specific task, which limits the application of GNN models. Secondly, existing GNN models do not make full use of the best architecture design practices in other established areas, e.g., computer vision (CV). For example, existing multi-layer GNN models tend to stack multiple layers with the same aggregator (see bottom left of Figure 1), which aggregates hidden features of multi-hop neighbors. However it remains to be seen whether combinations of different aggregators in a multi-layer GNN model can further improve the performance. These challenges lead to a straightforward question: *can we obtain well-behaved data-specific GNN architectures*?

To address the above question, researchers turn to neural architecture search (NAS) [14, 15, 16, 17] approaches, which have shown promising results in automatically designing architectures for convolutional neural networks (CNN) and recurrent neural networks (RNN). In very recent time, there are two preliminary works, GraphNAS [18] and Auto-GNN [19], making the first attempt to apply NAS to GNN architecture design. Though GraphNAS and Auto-GNN show some promising results, there are some drawbacks in expressive capability and search efficiency of GraphNAS and Auto-GNN due to the designed search space. In the NAS literature [14, 15, 16, 17], a good search space should be both expressive and compact. That is the search space should be large enough to subsume existing human-design architectures, thus the performance of a search method can be guaranteed. However, it will be extremely costly if the search space is too general, which is impractical for any searching method. The search spaces of GraphNAS and Auto-GNN are the same, both of which do not well satisfy the requirement of a good search space. On one hand, they fail to include several latest GNN models, e.g., the GeniePath [6], for which we give a more detailed analysis in Section 3.1 (Table 1). On the other hand, the search space includes too many choices, making it too complicated to search efficiently.

In this work, to overcome the drawbacks of GraphNAS and Auto-GNN and push forward the research of NAS approaches for GNN, we propose the SNAG framework (Simplified Neural Architecture search for Graph neural networks), consisting of a simpler yet more expressive search space and a RL-based search algorithm. By revisiting extensive existing works, we unify state-of-the-art GNN models in a message passing framework [7], based on which a much more expressive yet simpler search space is designed. The simplified search space can not only emulate a series of existing GNN models, but also be very flexible to use the weight sharing mechanism, which is a widely used technique to accelerate the search algorithm in the NAS literature. We conduct

Proceedings of the CIKM 2020 Workshops, October 19-20, Galway, Ireland.

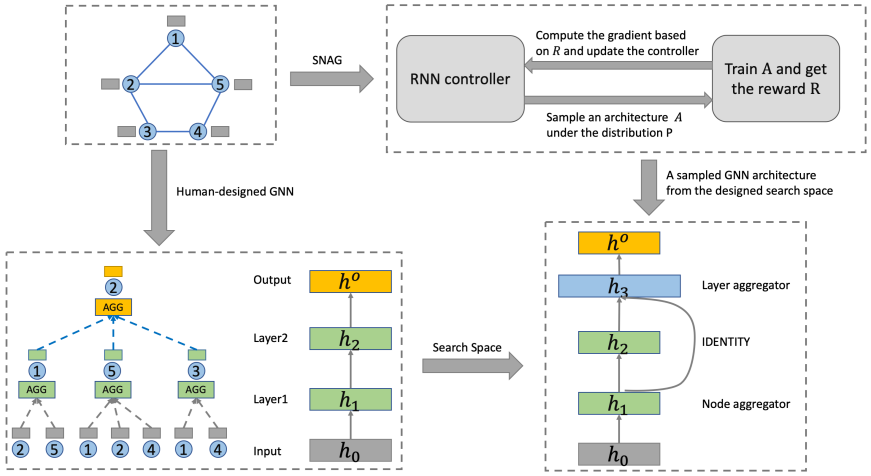


Figure 1: The whole framework of the propsed SNAG (Best view in color). (a) Upper Left: an example graph with five nodes. The gray rectangle represent the input features attached to each node; (b) Bottom Left: a typical 2-layer GNN architecture following the message passing neighborhood aggregation schema, which computes the embeddings of node "2"; (c) Upper Right: the reinforcement learning pipeline for NAS; d) Bottom Right: an illustration of a search space of the proposed SNAG using 2-layer GNN as backbone, which includes two key components of existing GNN models: node and layer aggregators.

extensive experiments to demonstrate the effectiveness of the SNAG framework comparing to various baselines including GraphNAS and Auto-GNN. To summarize, the contributions of this work are in the following:

- In this work, to automatically obtain well-behaved data-specific GNN architectures, we propose the SNAG framework, which can overcome the drawbacks of existing NAS approaches, i.e., GraphNAS and Auto-GNN. To better utilize the NAS techniques, we design a novel and effective search space, which can emulate more existing GNN architectures than previous works.
- We design a RL-based search algorithm and its variant by adopting the weight sharing mechanism (SNAG-WS). By comparing the performance of these two variants, we show that the weight sharing mechanism is not empirically useful as we imagined, which aligns with the latest research in NAS literature [20].
- Extensive experiments on real-world datasets are conducted to evaluate the proposed SNAG framework, comparing to human-designed GNNs and NAS methods. The experimental results demonstrate the superiority of SNAG in terms of effectiveness and efficiency compared extensive baseline models.

2. Related Works

2.1. Graph Neural Network (GNN)

GNN is first proposed in [1] and in the past five years many different variants [8, 9, 10, 11, 12, 13, 6] have been designed, all of which are relying on a neighborhood aggregation (or *message passing*) schema [7]. As shown in the left part of Figure 1, it tries to learn the representation of a given node in a graph by iteratively aggregating the hidden features ("message") of its neighbors, and the message can propagate to farther neighborhood in the graph, e.g., the hidden features of two-hop neighbors can be aggregated in a two-step iteration process. Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be a simple graph with node features $\mathbf{X} \in \mathbb{R}^{N \times d}$, where $\mathcal{V}$ and $\mathcal{E}$ represent the node and edge sets, respectively. N represents the number of nodes and d is the dimension of node features. We use $N(v)$ to represent the first-order neighbors of a node v in $\mathcal{G}$, i.e., $N(v) = \{u \in \mathcal{V} | (v, u) \in \mathcal{E}\}$. In the literature, we also create a new set $\widetilde{N}(v)$ is the neighbor set including itself, i.e., $\widetilde{N}(v) = \{v\} \cup \{u \in \mathcal{V} | (v, u) \in \mathcal{E}\}$.

Then a K-layer GNN can be written as follows: the l-th layer ($l = 1, \cdots, K$) updates $\mathbf{h}_v$ for each node v by aggregating its neighborhood as

$$\mathbf{h}_v^l = \sigma\left(\mathbf{W}^{(l)} \cdot \Phi_n\left(\{\mathbf{h}_u^{(l-1)}, \forall u \in \widetilde{N}(v)\}\right)\right), \quad (1)$$

where $\mathbf{h}_v^{(l)} \in \mathbb{R}^{d_l}$ represents the hidden features of a node v learned by the l-th layer, and d_l is the

Table 1
Comparisons of the search space between existing NAS methods and SNAG. For more details of the "Others" columns of GraphNAS/Auto-GNN, we refer readers to the corresponding papers.

	Node aggregators	Layer aggregators	Others
GraphNAS/ Auto-GNN	GCN,SAGE-SUM/-MEAN/-MAX, MLP, GAT , GAT-SYM/-COS/ -LINEAR/-GEN-LINEAR ,	-	Hidden Embedding Size, Attention Head, Activation Function
Ours	All above plus SAGE-LSTM and GeniePath	CONCAT,MAX,LSTM	IDENTITY, ZERO

corresponding dimension. $\mathbf{W}^{(l)}$ is a trainable weight matrix shared by all nodes in the graph, and σ is a non-linear activation function, e.g., a sigmoid or ReLU. Φ_n is the key component, i.e., a pre-defined aggregation function, which varies across on different GNN models. For example, in [8], a weighted summation function is designed as the node aggregators, and in [9], different functions, e.g., mean and max pooling, are proposed as the aggregators. Further, to weigh the importance of different neighbors, attention mechanism is incorporated to design the aggregators [10].

Usually, the output of the last layer is used as the final representation for each node, which is denoted as $\mathbf{z}_v = \mathbf{h}_v^{(K)}$. In [12], skip-connections [21] are incorporated to propagate message from intermediate layers to an extra layer, and the final representation of the node v is computed by a layer aggregation as $\mathbf{z}_v = \Phi_l\left(\mathbf{h}_v^{(1)}, \cdots, \mathbf{h}_v^{(K)}\right)$, and Φ_l can also have different options, e.g., max-pooling, concatenation. Based on the node and layer aggregators, we can define the two key components of exiting GNN models, i.e., the neighborhood aggregation function and the range of the neighborhood, which tends to be tuned depending on the tasks. In Table 1, we list all node and layer aggregators in this work, which lays the basis for the proposed SNAG framework.

2.2. Neural Architecture Search (NAS)

Neural architecture search (NAS) [14, 15, 16, 17] aims to automatically find better and smaller architectures comparing to expert-designed ones, which have shown promising results in architecture design for CNN and Recurrent Neural Network (RNN) [22, 23, 24, 25, 26]. In the literature, one of the representative NAS approaches are reinforcement learning (RL) [14, 15, 27], which trains an RNN controller in the loop: the controller firstly generates an candidate architecture by sampling a list of actions (operations) from a pre-defined search space, and then trains it to convergence to obtain the performance of the given task. The controller then uses the performance as the guiding signal to update the RNN parameters, and the whole process is repeated for many iterations to find more promising architectures. GraphNAS [18] and Auto-GNN [19] are the first two RL-based NAS methods for GNN.

Search space is a key component of NAS approaches, the quality of which directly affects the final performance and search efficiency. As mentioned in [14, 15, 23, 28, 27, 22, 29, 26, 25], a good search space should include existing human-designed models, thus the performance of an designed search algorithm can be guaranteed. In this work, by unifying existing GNN models in the message passing framework [7] with the proposed node and layer aggregators, we design a more expressive yet simple search space in this work, which is also flexible enough to incorporate the weight sharing mechanism into our RL-based method.

3. The Proposed Framework

3.1. The design of search space

As introduced in Section 2.1, most existing GNN architectures are relying on a message passing framework [7], which constitutes the backbone of the designed search space in this work. Besides, motivated by JK-Network [13], to further improve the expressive capability, we modify the message framework by adding an extra layer which can adaptively combine the outputs of all node aggregation layers. In this work, we argue and demonstrate in the experiments that these two components are the key parts for a well-behaved GNN model, denoted as *Node arggregators* and *Layer aggregators*. The former one focus on how to aggregate the neighborhood features, while the latter one focus on the range of neighborhood to use. Here we introduce the backbone of the proposed search space, as shown in the bottom right part of Figure 1, which consists of two key components:

- **Node aggregators**: We choose 12 node aggregators based on popular GNN models, and they are presented in Table 1.
- **Layer aggregators**: We choose 3 layer aggregators as shown in Table 1. Besides, we have two more

operations, IDENTITY and ZERO, related to skip-connections. Instead of requiring skip-connections between all intermediate layers and the final layer in JK-Network, in this work, we generalize this option by proposing to search for the existence of skip-connection between each intermediate layer and the last layer. To connect, we choose IDENTITY, and ZERO otherwise.

To further inject the domain knowledge from existing GNN architectures, when searching for the skip-connections for each GNN layer, we add one more constraint that the last layer should always be used as the final output, thus for a K-layer GNN architecture, we need to search $K-1$ IDENTITY or ZERO for the skip-connection options.

3.2. Problem formulation

After designing the search space, denoted as $\mathcal{A}$, the search process implies a bi-level optimization problem [30, 31], as show in the following:

$$\begin{aligned} \min_{\alpha \in \mathcal{A}} \quad & \mathcal{L}_{val}(\alpha, w^*), \\ \text{s.t.} \quad & w^* = \arg\min_w \mathcal{L}_{train}(\alpha, w), \end{aligned} \tag{2}$$

where $\mathcal{L}_{train}$ and $\mathcal{L}_{val}$ represent the training and validation loss, respectively, and $\mathcal{A}$ represents the search space introduced in Section 3.1. α and w represent the architecture and model parameters. Eq. (2) denotes a trial-and-error process for the NAS problem, which selects an architecture α from the search space, and then trains it from scratch to obtain the best performance. This process is repeated during the given time budget and the optimal α^* is kept track of and returned after the search process finished.

In this work, motivated by the pioneering NAS works [14, 15], we design a RL method to execute the search process. To be specific, during the search phase, we use a recurrent neural network (RNN) controller, parameterized by θ_c, to sample an candidate architecture from the search space. The architecture is represented by a list of actions (OPs), including the node aggregators, layer aggregators and IDENTITY/ZERO as shown in Table 1. Then the candidate architecture will be trained till convergence, and the accuracy on a held-out validation set $\mathcal{D}_{val}$ is returned. The parameters of the RNN controller are then optimized in order to maximize the expected validation accuracy $\mathbb{E}_{P(\alpha;\theta_c)}[\mathcal{R}]$ on $\mathcal{D}_{val}$, where $P(\alpha;\theta_c)$ is the distribution of architectures parameterized by θ_c, and $\mathcal{R}$ is the validation accuracy. In this way, the RNN controller will generate better architectures over time, and can obtain optimal one in the end of the search phase. After finishing the search process, we need to derive the searched architectures. We first sample n architectures under the trained distribution $P(\alpha, \theta_c)$, and for each architecture, we train them from scratch with some hyper-parameters tuning, e.g., the embedding size and learning rate, etc. We then select the best architecture as the searched one, which aligns with the process in previous works [15, 27]. In our experiments, we empirically set $n = 10$ for simplicity. For more technical details, we refer readers to [15, 18].

Besides, in this work, we also incorporate the weight sharing mechanism into our framework, and propose the SNAG-WS variant. The key difference between SNAG and SNAG-WS lies in that we create a dictionary to load and save the trained parameters of all OPs (Table 1) in a sampled architecture during the search process.

Table 2
Dataset statistics of the datasets in the experiments.

	Transductive			Inductive
	Cora	CiteSeer	PubMed	PPI
#nodes	2,708	3,327	19,717	56,944
#edges	5,278	4,552	44,324	818,716
#features	1,433	3,703	500	121
#classes	7	6	3	50

4. Experiments

4.1. Experimental Settings

4.1.1. Datasets and Tasks.

Here, we introduce two tasks and the corresponding datasets (Table 2), which are standard ones in the literature [8, 9, 13].

Transductive Task. Only a subset of nodes in one graph are used as training data, and other nodes are used as validation and test data. For this setting, we use three benchmark dataset: Cora, CiteSeer, PubMed. They are all citation networks, provided by [32]. Each node represents a paper, and each edge represents the citation relation between two papers. The datasets contain bag-of-words features for each paper (node), and the task is to classify papers into different subjects based on the citation networks.

For all datasets, We split the nodes in all graphs into 60%, 20%, 20% for training, validation, and test. For the transductive task, we use the classification accuracy as the evaluation metric.

Inductive Task. In this task, we use a number of graphs as training data, and other completely unseen graphs as validation/test data. For this setting, we use the PPI dataset, provided by [9], on which the task is to classify protein functions. PPI consists of 24 graphs, with each corresponding to a human tissue. Each node has positional gene sets, motif gene sets and immunological

Table 3
Performance comparisons in transductive tasks. We show the mean classification accuracy (with standard deviation). We categorize baselines into human-designed GNNs and NAS methods. The best results in different groups of baselines are underlined, and the best result on each dataset is in boldface.

		Transductive		
	Methods	Cora	CiteSeer	PubMed
Human-designed GNN	GCN	0.8761 (0.0101)	0.7666 (0.0202)	0.8708 (0.0030)
	GCN-JK	0.8770 (0.0118)	0.7713 (0.0136)	0.8777 (0.0037)
	GraphSAGE	0.8741 (0.0159)	0.7599 (0.0094)	0.8784 (0.0044)
	GraphSAGE-JK	0.8841 (0.0015)	0.7654 (0.0054)	0.8822 (0.0066)
	GAT	0.8719 (0.0163)	0.7518 (0.0145)	0.8573 (0.0066)
	GAT-JK	0.8726 (0.0086)	0.7527 (0.0128)	0.8674 (0.0055)
	GIN	0.8600 (0.0083)	0.7340 (0.0139)	0.8799 (0.0046)
	GIN-JK	0.8699 (0.0103)	0.7651 (0.0133)	0.8828 (0.0054)
	GeniePath	0.8670 (0.0123)	0.7594 (0.0137)	0.8796 (0.0039)
	GeniePath-JK	0.8776 (0.0117)	0.7591 (0.0116)	0.8818 (0.0037)
	LGCN	0.8687 (0.0075)	0.7543 (0.0221)	0.8753 (0.0012)
NAS methods	Random	0.8694 (0.0032)	**0.7820 (0.0020)**	0.8888(0.0009)
	Bayesian	0.8580 (0.0027)	0.7650 (0.0021)	0.8842(0.0005)
	GraphNAS	0.8840 (0.0071)	0.7762 (0.0061)	0.8896 (0.0024)
	GraphNAS-WS	0.8808 (0.0101)	0.7613 (0.0156)	0.8842 (0.0103)
ours	SNAG	0.8826 (0.0023)	0.7707 (0.0064)	0.8877 (0.0012)
	SNAG-WS	**0.8895 (0.0051)**	0.7695 (0.0069)	**0.8942 (0.0010)**

Table 4
Performance comparisons in inductive tasks. We show the Micro-F1 (with standard deviation). We categorize baselines into human-designed GNNs and NAS methods. The best results in different groups of baselines are underlined, and the best result is in boldface.

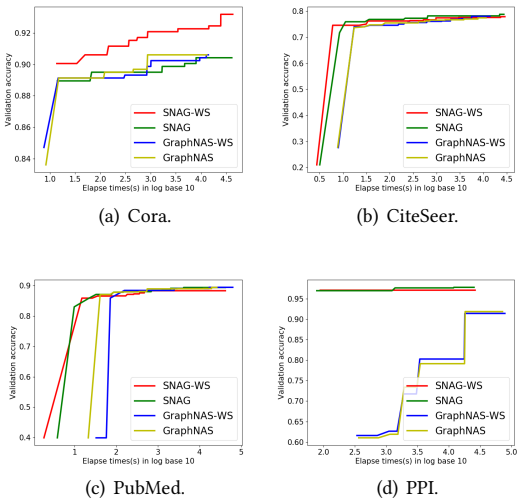
	Methods	PPI
Human-designed GNN	GCN	0.9333 (0.0019)
	GCN-JK	0.9344 (0.0007)
	GraphSAGE	0.9721 (0.0010)
	GraphSAGE-JK	0.9718 (0.0014)
	GAT	0.9782 (0.0005)
	GAT-JK	0.9781 (0.0003)
	GIN	0.9593 (0.0052)
	GIN-JK	0.9641 (0.0029)
	GeniePath	0.9528 (0.0000)
	GeniePath-JK	0.9644 (0.0000)
NAS methods	Random	0.9882 (0.0011)
	Bayesian	**0.9897 (0.0008)**
	GraphNAS	0.9698 (0.0128)
	GraphNAS-WS	0.9584 (0.0415)
ours	SNAG	**0.9887 (0.0010)**
	SNAG-WS	0.9875 (0.0006)

signatures as features and gene ontology sets as labels. 20 graphs are used for training, 2 graphs are used for validation and the rest for testing, respectively. For the inductive task, we use Micro-F1 as the evaluation metric.

4.1.2. Compared Methods

We compare SNAG with two groups of state-of-the-art methods: human-designed GNN architectures and NAS methods for GNN.

Human-designed GNNs. We use the following popular GNN architectures: GCN [8], GraphSAGE [9], GAT [10],



(a) Cora. (b) CiteSeer.

(c) PubMed. (d) PPI.

Figure 2: The validation accuracy w.r.t. search time (in seconds) in log base 10.

GIN [12], LGCN [11], GeniePath [6]. For models with variants, like different aggregators in GraphSAGE or different attention functions in GAT, we report the best performance across the variants. Besides, we extend the idea of JK-Network [13] in all models except for LGCN, and obtain 5 more baselines: GCN-JK, GraphSAGE-JK, GAT-JK, GIN-JK, GeniePath-JK, which add an extra layer. For LGCN, we use the code released by the authors [1]. For other baselines, we use the popular open source library Pytorch Geometric (PyG) [33] [2], which implements various GNN models. For all baselines, we train it from scratch with the obtained best hyper-parameters on validation datasets, and get the test performance. We repeat this process for 5 times, and report the final mean accuracy with standard deviation.

NAS methods for GNN. We consider the following methods: Random search (denoted as "Random") and Bayesian optimization [34] (denoted as "Bayesian"), which directly search on the search with random sampling and bayesian optimization methods, respectively. Besides, GraphNAS[3] [18] is chosen as NAS baseline.

Note that for human-designed GNNs and NAS methods, for fair comparison and good balance between efficiency and performance, we choose set the number of GNN layers to be 3, which is an empirically good choice in the literature [10, 6].

[1] https://github.com/HongyangGao/LGCN
[2] https://github.com/rusty1s/pytorch_geometric
[3] https://github.com/GraphNAS/GraphNAS

4.2. Performance comparison

In this part, we give the analysis of the performance comparisons on different datasets.

From Table 3, we can see that SNAG models, including SNAG-WS, win over all baselines on most datasets except CiteSeer. Considering the fact that the performance of SNAG on CiteSeer is very close to the best one (Random), it demonstrates the effectiveness of the NAS methods on GNN. In other words, with SNAG, we can obtain well-behaved GNN architectures given a new task. When comparing SNAG with GraphNAS methods, the performance gain is evident. We attribute this to the superiority of the expressive yet simple search space.

From Table 4, we can see that the performance trending is very similar to that in transductive task, which is that the NAS methods can obtain better or competitive performance than human-designed GNNs. When looking at the NAS methods, we can see that our proposed SNAG, Random and Bayesian outperforms GraphNAS. This also demonstrates the superiority of the designed search space.

Taking into consideration the results of these two tasks, we demonstrate the effectiveness of SNAG models, especially the superiority of the search space.

4.3. Understanding the search space of SNAG

In this section, we show the simplicity and expressiveness of the designed search space of SNAG from two aspects: speedup in searching and the performance gain from the layer aggregators.

4.3.1. Speedup in searching

In this part, to show the simplicity of the designed search space, we compare the efficiency of SNAG and GraphNAS by showing the validation accuracy w.r.t to the running time, and the results are shown in Figure 2. The accuracy is obtained by evaluating the sampled architecture on validation set after training it from scratch till convergency, which can reflect the capability of NAS methods in discovering better architectures with time elapsing. From Figure 2, we can see that SNAG speeds up the search process significantly comparing to GraphNAS, i.e., the model can obtain better GNN architectures during the search space. Considering the fact that both GraphNAS and SNAG adopt the same RL framework, then this advantage is attributed to simpler and smaller search space.

4.3.2. Influence of layer aggregators

In this part, to show the stronger expressive capability of the designed search space, we conduct experiments

Table 5
Performance comparisons of SNAG and SNAG-WS using different search spaces.

	SNAG		SNAG-WS	
	layer aggregators (w)	layer aggregators (w/o)	layer aggregators (w)	layer aggregators (w/o)
Cora	0.8826 (0.0023)	0.8822 (0.0071)	0.8895 (0.0051)	0.8892 (0.0062)
CiteSeer	0.7707 (0.0064)	0.7335 (0.0025)	0.7695 (0.0069)	0.7530 (00034)
PubMed	0.8877 (0.0012)	0.8756 (0.0016)	0.8942 (0.0010)	0.8800 (0.0013)
PPI	0.9887 (0.0010)	0.9849 (0.0040)	0.9875 (0.0006)	0.9861 (0.0009)

on all datasets using a search space only with the node aggregators, i.e., removing the layer aggregators, as comparisons. The results are shown in Table 5, and we report the test accuracies of both the SNAG and SNAG-WS. From Table 5, we can see that the performance consistently drops on all datasets when removing the layer aggregators, which demonstrates the importance of the layer aggregators for the final performance and aligns with the observation in Section 4.2 that the performance of human-designed GNNs can be improved by adopting the JK-Network architecture.

5. Conclusion and Future work

In this work, to overcome the drawbacks in expressive capability and search efficiency of two existing NAS approaches for GNN, i.e., GraphNAS [18] and Auto-GNN [19], we propose the SNAG framework, i.e., Simplified Neural Architecture search for GNN. By revisiting existing works, we unify state-of-the-art GNN models in a message passing framework [7], and design a simpler yet more expressive search space than that of GraphNAS and Auto-GNN. A RL-based search algorithm is designed and a variant (SNAG-WS) is also proposed by incorporating the weight sharing mechanism. Through extensive experiments on real-world datasets, we not only demonstrate the effectiveness of the proposed SNAG framework comparing to various baselines including GraphNAS and Auto-GNN, but also give better understanding of different components of the proposed SNAG. For future work, we will explore the SNAG framework in more graph-based tasks besides node classification.

References

[1] M. Gori, G. Monfardini, F. Scarselli, A new model for learning in graph domains, in: IJCNN, volume 2, 2005, pp. 729–734.

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

[3] R. Ying, R. He, K. Chen, P. Eksombatchai, W. L. Hamilton, J. Leskovec, Graph convolutional neural networks for web-scale recommender systems, in: KDD, 2018, pp. 974–983.

[4] J. Wang, P. Huang, H. Zhao, Z. Zhang, B. Zhao, D. L. Lee, Billion-scale commodity embedding for e-commerce recommendation in alibaba, in: KDD, 2018, pp. 839–848.

[5] W. Xiao, H. Zhao, H. Pan, Y. Song, V. W. Zheng, Q. Yang, Beyond personalization: Social content recommendation for creator equality and consumer satisfaction, in: KDD, 2019, pp. 235–245.

[6] Z. Liu, C. Chen, L. Li, J. Zhou, X. Li, L. Song, Y. Qi, Geniepath: Graph neural networks with adaptive receptive paths, in: AAAI, volume 33, 2019, pp. 4424–4431.

[7] J. Gilmer, S. S. Schoenholz, P. F. Riley, O. Vinyals, G. E. Dahl, Neural message passing for quantum chemistry, in: ICML, 2017, pp. 1263–1272.

[8] T. N. Kipf, M. Welling, Semi-supervised classification with graph convolutional networks, ICLR (2016).

[9] W. Hamilton, Z. Ying, J. Leskovec, Inductive representation learning on large graphs, in: NeurIPS, 2017, pp. 1024–1034.

[10] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Lio, Y. Bengio, Graph attention networks, ICLR (2018).

[11] H. Gao, Z. Wang, S. Ji, Large-scale learnable graph convolutional networks, in: KDD, 2018, pp. 1416–1424.

[12] K. Xu, W. Hu, J. Leskovec, S. Jegelka, How powerful are graph neural networks?, in: ICLR, 2019.

[13] K. Xu, C. Li, Y. Tian, T. Sonobe, K.-i. Kawarabayashi, S. Jegelka, Representation learning on graphs with jumping knowledge networks, in: ICML, 2018, pp. 5449–5458.

[14] B. Baker, O. Gupta, N. Naik, R. Raskar, Designing neural network architectures using reinforcement learning, ICLR (2017).

[15] B. Zoph, Q. V. Le, Neural architecture search with reinforcement learning, ICLR (2017).

[16] T. Elsken, J. H. Metzen, F. Hutter, Neural architecture search: A survey, JMLR (2018).
[17] Q. Yao, M. Wang, Taking human out of learning applications: A survey on automated machine learning, arXiv preprint arXiv:1810.13306 (2018).
[18] Y. Gao, H. Yang, P. Zhang, C. Zhou, Y. Hu, Graph neural architecture search, in: IJCAI, 2020, pp. 1403–1409.
[19] K. Zhou, Q. Song, X. Huang, X. Hu, Auto-GNN: Neural Architecture Search of Graph Neural Networks, Technical Report, arXiv preprint arXiv:1909.03184, 2019.
[20] C. Sciuto, K. Yu, M. Jaggi, C. Musat, M. Salzmann, Evaluating the search phase of neural architecture search, ICLR (2020).
[21] K. He, X. Zhang, S. Ren, J. Sun, Deep residual learning for image recognition, in: CVPR, 2016, pp. 770–778.
[22] H. Liu, K. Simonyan, Y. Yang, Darts: Differentiable architecture search, ICLR (2019).
[23] B. Zoph, V. Vasudevan, J. Shlens, Q. V. Le, Learning transferable architectures for scalable image recognition, in: CVPR, 2018, pp. 8697–8710.
[24] M. Tan, Q. Le, Efficientnet: Rethinking model scaling for convolutional neural networks, in: ICML, 2019, pp. 6105–6114.
[25] Q. Yao, J. Xu, W.-W. Tu, Z. Zhu, Efficient neural architecture search via proximal iterations., in: AAAI, 2020, pp. 6664–6671.
[26] Y. Zhang, Q. Yao, L. Chen, Neural Recurrent Structure Search for Knowledge Graph Embedding, Technical Report, International Workshop on Knowledge Graph@KDD, 2019.
[27] H. Pham, M. Guan, B. Zoph, Q. Le, J. Dean, Efficient neural architecture search via parameter sharing, in: ICML, 2018, pp. 4092–4101.
[28] G. Bender, P. Kindermans, B. Zoph, V. Vasudevan, Q. V. Le, Understanding and simplifying one-shot architecture search, in: ICML, 2018, pp. 549–558.
[29] L. Li, A. Talwalkar, Random search and reproducibility for neural architecture search, arXiv preprint arXiv:1902.07638 (2019).
[30] B. Colson, P. Marcotte, G. Savard, An overview of bilevel optimization, Annals of operations research 153 (2007) 235–256.
[31] L. Franceschi, P. Frasconi, S. Salzo, R. Grazzi, M. Pontil, Bilevel programming for hyperparameter optimization and meta-learning, in: ICML, 2018, pp. 1568–1577.
[32] P. Sen, G. Namata, M. Bilgic, L. Getoor, B. Galligher, T. Eliassi-Rad, Collective classification in network data, AI magazine 29 (2008) 93–93.
[33] M. Fey, J. E. Lenssen, Fast graph representation learning with PyTorch Geometric, in: ICLRW, 2019.
[34] J. S. Bergstra, R. Bardenet, Y. Bengio, B. Kégl, Algorithms for hyper-parameter optimization, in: NeurIPS, 2011, pp. 2546–2554.