A DETAILS OF EXPERIMENT SETTINGS

A.1 The implementation details of PAS

All models are implemented with Pytorch [31] on a GPU 2080Ti (Memory: 12GB, Cuda version: 10.2). Thus, for consistent comparisons of baseline models, we use the implementation of all GNN baselines by the popular GNN library: Pytorch Geometric (PyG) (version 1.6.1) [9], which provides a unifying code framework [4] for various GNN models. Further, we adopt the same data-preprocessing manner by PyG[5] and split data by means of a stratification technique with the same seed.

For all human-designed global and hierarchical pooling baselines, we search the layer numbers of this method, global pooling functions, embedding size, dropout rate, and learning rate as shown in Table 6. Following the DiffPool [49], we set the pooling rate $k = \frac{L}{10} \times N$ in Eq. (3) for all pooling operations where L is the layer number of this method and N is the node number. For each model, select 30 hyperparameters settings with Hyperopt [6], and evaluate each setting on 10-fold cross-validation data. We select settings based on mean validation accuracy then report the final test accuracy and the standard deviations. Besides, we also provide the comparisons of baseline reported results in Table 7 to show the influence of different settings.

We set training and finetuning stages to get the 10-fold cross-validation test accuracy for all NAS baselines and PAS in this paper. In the training stage, the dataset is split into 80% for training, 10% for validation and test with the same seed, and we select architectures from the supernet as shown in Alg. 1. In the finetuning stage, we tune these searched architectures over the pre-defined space as shown in Table 8 with Hyperopt based on 10-fold data, and select the final candidate with mean validation accuracy.

For all RL-based methods, GraphNAS [7], GraphNAS-WS, SNAG [8] and SNAG-WS, we set the training epoch to 200. In each training epoch, we sample 10 architectures and use the validation accuracy to update the controller parameters. After training finished, we sample 5 candidates with the controller.

For EA baseline, we follow the experiments in [15][9]. We set the population size to 50 and the training epoch to 40. In each training epoch, random select an architecture and mutates all operations with probability 0.1 to generate the new architecture; random select two architectures and crossed to generate one new architecture, 25 mutation operations and 25 crossover operations in each training epoch. The architecture is derived based on the validation accuracy after the training stage terminates.

For Random and Bayesian methods, we set the training epoch to 200. In each training epoch, sample one architecture and train from scratch. After training finished, we select one candidate with the validation accuracy.

For PAS, we set the training epoch to 200 as shown in Alg. 1. In each training epoch, PAS samples a set of minibatchs and uses the training loss to update parameters $\mathbf{W}$ and use the validation loss to update $\boldsymbol{\alpha}$. After search process is finished, we derive the candidate architecture from the supernet. Repeat 5 times with different seeds, we can get 5 candidates.

In the finetuning stage, each candidate architecture owns 30 hyper steps. In each hyper step, a set of hyperparameters will be sampled from Table 8 based on Hyperopt, then we generate final performance on 10-fold data. We choose the hyperparameters for each candidate with the mean validation accuracy. After that, we choose the candidate with the mean validation accuracy then report the final test accuracy and the standard deviations based on 10-fold cross-validation data.

Table 6: Hyperparameter space for human-designed baselines.

Dimension	Operation
Layer number	1, 2, 3, 4, 5
Global pooling function	`GLOBAL_MEAN`, `GLOBAL_SUM`
Embedding size	8, 16, 32, 64, 128, 256, 512
Dropout rate	0, 0.1, 0.2,...,0.9
Learning rate	[0.001, 0.025]

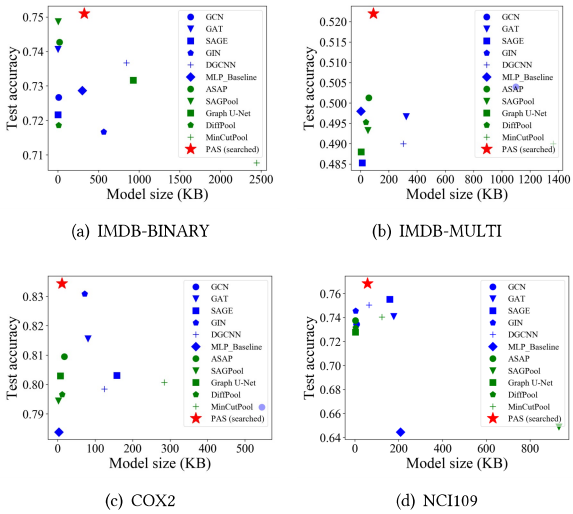
Table 7: The reported results of 3 methods (denoted as Method1) in other methods (denoted as Method2).

Method 1	Method 2	D&D	PROTEINS
DiffPool [49]	FAIR [8]	0.7500	0.7370
	SAGPool [23]	0.6695	0.6820
	DiffPool [49]	0.8064	0.7625
	Graph U-Net [11]	0.8064	0.7625
	ASAP [34]	0.6695	0.6820
	PAS	0.7775	0.7355
DGCNN [53]	FAIR [8]	0.7660	0.7290
	DGCNN [53]	0.7937	0.7554
	Graph U-Net [11]	0.7937	0.7626
	DiffPool [49]	0.7937	0.7554
	SAGPool [23]	0.7253	0.6672
	ASAP [34]	0.7187	0.7391
	PAS	0.7666	0.7357
Graph U-Net [11]	ASAP [34]	0.7501	0.7110
	SAGPool [23]	0.7501	0.7110
GraphSAGE [16]	FAIR [8]	0.7290	0.7300
	DiffPool [49]	0.7542	0.7048
	PAS	0.7727	0.7375

Table 8: Hyperparameter space in the finetuning stage for NAS methods.

Dimension	Operation
Embedding size	8, 16, 32, 64, 128, 256
Dropout rate	0, 0.1, 0.2, $\cdots$, 0.9
Learning rate	[0.001, 0.025]
Optimizer	Adam, AdaGrad
Activation function	RELU, ELU

[4] https://github.com/rusty1s/pytorch_geometric/tree/master/benchmark/kernel
[5] https://github.com/rusty1s/pytorch_geometric/blob/master/benchmark/kernel/datasets.py
[6] https://github.com/hyperopt/hyperopt
[7] https://github.com/GraphNAS/GraphNAS
[8] https://github.com/AutoML-Research/SNAG
[9] https://github.com/megvii-model/SinglePathOneShot



(a) IMDB-BINARY (b) IMDB-MULTI

(c) COX2 (d) NCI109

Figure 9: (Best viewed in color) The test accuracy w.r.t. model size. The searched architectures by PAS can achieve SOTA performance with moderate size in terms of the model parameters.

Table 9: The search space we use in GraphGym. Other parameters remain the same.

Pre-process layer	1, 2
Message Passing layers	1, 2, 3, 4, 5, 6
Post-process layers	1, 2
Pooling layer	SAGPOOL, TOPKPOOL, ASAP, NONE
Readout layer	GLOBAL_MEAN, GLOBAL_MAX, GLOBAL_SUM, GLOBAL_ATT
Learning rate	0.01
Batch size	16
Training epochs	100

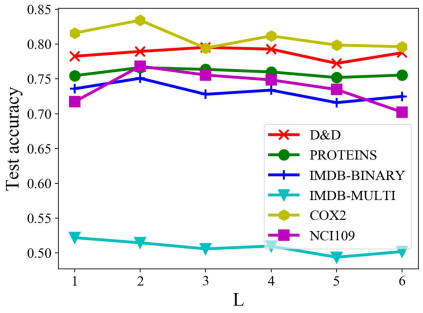


Figure 8: The test accuracy w.r.t. the layer numbers for PAS.

A.2 The implementation details of GraphGym

In this section, we show the details of the designed experiments as mentioned in Figure 1.

Vere recently, GraphGym [51] was proposed to evaluate the design dimensions of GNN models, like aggregation functions, number of layers, etc. However, for the graph classification task, GraphGym only uses a fixed global pooling function to generate the graph representations based on node aggregation operations. It cannot evaluate different pooling methods for graph classification. To further evaluate the pooling methods, we design a search space for graph classification on top of GraphGym. As shown in Figure 6, we add the pooling layer behind the 1-st, 3-rd, 5-th GNN layer, and add one Readout layer in the post-process stage. The search dimensions are shown in Table 9.

We use the 6 real-world datasets and 8 synthetic datasets mentioned in GraphGym. Each synthetic dataset chooses graph structure function from {scalefree, smallworld} and choose feature function from {clustering, const, onehot, pagerank}. 256 graphs are generated with different Average Clustering Coefficient and Average Path Length. We use 420 setups on these 14 datasets so that each dataset has 30 hits on average, and the results are shown in Figure 1 and 7.

B EXPERIMENTS

B.1 More figures

For sake of the space, we only show the test accuracy and model size comparisons on 2 datasets in Figure 4. Here, more figures are shown in Figure 9.

B.2 The influence of layers

Here we conduct experiments to show the influences of the layer number L of PAS by varying $L \in \{1, 2, 3, 4, 5, 6\}$ in PAS. As introduced in Section 3.2 and Figure 2(b), each layer consists of one Aggregation and Pooling Module in PAS. The results are shown in Figure 8, from which we can see that the trending that the performance increases firstly with the increase of L and then decreases. Looking back at the statistics of the datasets in Table 3, we can get an empirical guideline for architecture design for graph classification, although it may not always be true, that the larger number of layers is preferable for graphs in the larger size (more nodes).