A Dataset Description

We evaluate our proposed GraphPatcher as well as other frameworks that mitigate the degree bias problem on seven real-worlds datasets spanning various fields such as citation network and merchandise network. Their statistics are shown in Table 4. For `Cora`, `Citeseer` and, `Pubmed`, we explore the community acknowledged public splits (i.e., fixed 20 nodes per class for training, 500 nodes for validation, and 1000 nodes for testing); whereas for `ogbn-arxiv`, we use the API from Open Graph Benchmark (OGB)[2] and explore the provided splits. For `Wiki.CS`, `Amazon-Photo`, and `Coauthor-CS`, we randomly select 10% nodes for training, another 10% for validation, and the remaining 80% for testing. We use the API from Deep Graph Library (DGL)[3] to load all datasets.

Table 4: Dataset Statistics.

Dataset	# Nodes	# Edges	# Features	Avg. Degree	Split
`Cora`	2,708	5,429	1,433	2.0	Public Split
`Citeseer`	3,327	4,732	3,703	1.4	Public Split
`Pubmed`	19,717	88,651	500	4.5	Public Split
`Wiki-CS`	11,701	216,123	300	18.5	10%/10%/80%
`Amazon-Photo`	7,650	119,043	745	15.6	10%/10%/80%
`Coauthor-CS`	18,333	81,894	6,805	4.5	10%/10%/80%
`ogbn-arxiv`	169,343	1,166,243	128	6.9	Public Split

B GraphPatcher Configuration and Experiment on Hyper-parameters

B.1 GraphPatcher Configuration

The architecture of GraphPatcher consists of two parts; the first part is a 2-layer GCN encoder that takes an ego-graph as input and vectorizes its nodes and the second part is an MLP that takes the representation of the anchor node and outputs the generated feature for the virtual patching node.

To ensure the reproducibility, we also provide the detailed hyper-parameter configurations of GraphPatcher for all datasets, as shown in Table 5. Besides, we use an early stopping strategy to decide the number of optimization steps, where the optimization stops if the validation loss stops decreasing for two consecutive steps.

Table 5: Hyper-parameters used for GraphPatcher.

Hyper-param.	`Cora`	`Citeseer`	`Pubmed`	`Wiki.CS`	`Am.Photo`	`Co.CS`	`Arxiv`
Augmentation strength	0.3	0.3	0.3	0.3	0.3	0.3	0.1
Patching step	3	3	3	3	3	3	5
# of sampled graphs	10 used for all datasets						
Batch size	64	64	64	8	16	4	16
Accumulation step	16	16	16	32	16	16	64
Learning rate	1e-4 used for all datasets						
Optimizer	AdamW with a weight decay of 1e-5 used for all datasets						

B.2 Experiment on Hyper-parameters

The hyper-parameters we tune for GraphPatcher include the number of patching nodes during the testing time, learning rate, hidden dimension, the augmentation strength at each step, and the total amount of patching steps. Experiments w.r.t. the number of patching nodes during the testing time has been showcased in Figure 3 and here we also append the results for the other four datasets, as shown in Figure 4. We observe similar trends as the aforementioned three datasets exhibit, where the

[2] https://ogb.stanford.edu

[3] https://www.dgl.ai

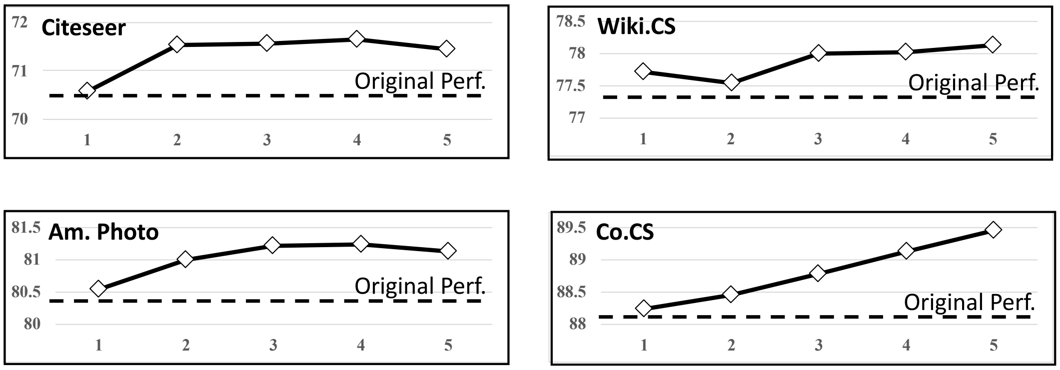

Figure 4: Overall perf. (y-axis) w.r.t. the number of patching nodes (x-axis).

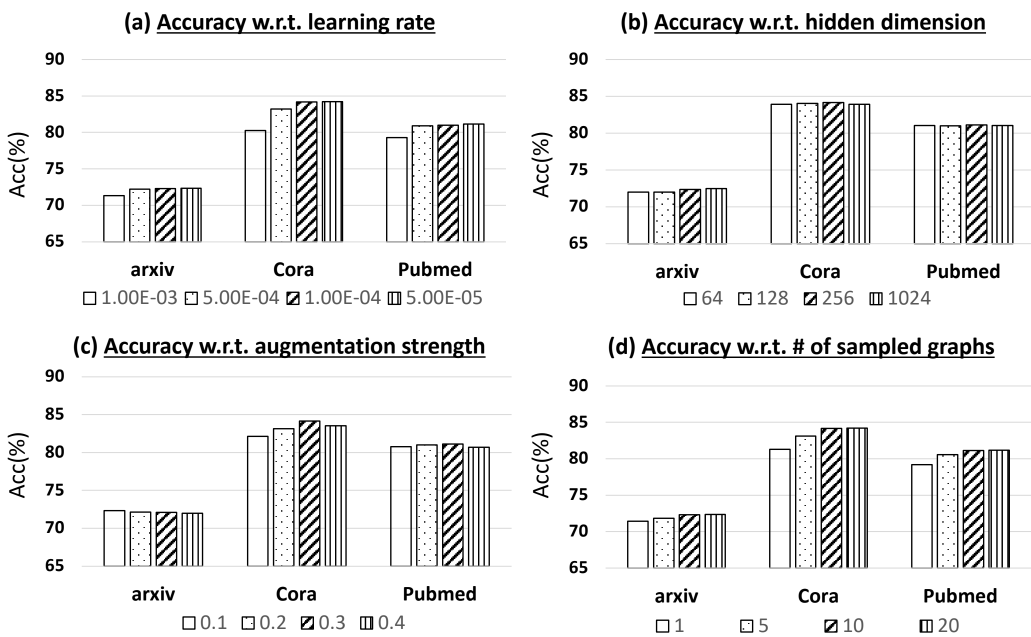

Figure 5: GRAPHPATCHER's sensitivity to different hyper-parameters.

performance of GRAPHPATCHER improves as the number of patching nodes increases and the gain saturates with 4 to 5 nodes patched.

We also conduct experiments w.r.t. learning rate, hidden dimension, the augmentation strength at each step, and the total amount of patching steps during the training. We tune the hidden dimension by conducting a grid search over common selections of [64, 128, 256, 1024] hidden units; we tune the learning rate similarly by searching over [1e-3, 5e-4, 1e-4, 5e-5]; and we tune the augmentation strength by searching over [0.1, 0.2, 0.3, 0.4].

The hidden dimension refers to the intermediate dimension of the 2-layer GCNs of GRAPHPATCHER. GRAPHPATCHER is constructed by a 2-layer GCN and features for virtual nodes are generated by a following multi-layer perceptron with the same hidden dimension. To reduce the search complexity, we explore an arithmetic sequence for the augmentation strength (i.e., the difference between any two consecutive strengths is the same) and set the total amount of patching steps during the training to $\lfloor \frac{1}{t} \rfloor$. For instance, an augmentation strength of 0.3 would lead to a 3-step training with augmentation strength of 0.3, 0.6, and 0.9 respectively. GRAPHPATCHER's sensitivity to these hyper-parameters is shown in Figure 5. Specifically, in Figure 5.(a) we can observe that across datasets, a large learning rate (i.e., 1e-3) leads to sub-optimal performance and GRAPHPATCHER achieves the best performance with a learning rate of 1e-4. We also investigate GRAPHPATCHER's sensitivity to the number of hidden dimensions (i.e., the model size). In Figure 5.(b), we notice that for large graphs like `Arxiv`, the performance gradually increases as the model size enlarges. And for small and medium graphs like `Cora` and `Pubmed`, the performance saturates with a hidden dimension of 128. Besides, in Figure 5.(c) we study GRAPHPATCHER's performance w.r.t. the augmentation strength (which can also be interpreted as the number of patching steps as described previously). We can observe that, for small and medium graphs, strong augmentation strength leads to better performance, due to the sparsity of the graph structures. Whereas for large graphs, small augmentation strength delivers good performance. Furthermore, to prove the effectiveness of our proposed training scheme with multiple ego-graphs, we train GRAPHPATCHER with different numbers of sampled graphs (i.e., L in Equation (5)), with the performance shown in Figure 5.(d). We can observe that without our proposed sampling strategy (i.e., the first column with $L = 1$), the performance of GRAPHPATCHER degrades significantly. As the number of sampled graphs gradually increases, the performance keeps improving and saturates with $L = 10$, empirically proving the effectiveness of the exploration of multiple ego-graphs for the same corruption strength.

B.3 Hardware and Software Configuration

We conduct experiments on a server having one RTX3090 GPU with 24 GB VRAM. The CPU we have on the server is an AMD Ryzen 3990X with 128GB RAM. The software we use includes DGL 1.9.0 and PyTorch 1.11.0. As for the baseline models that we compare GRAPHPATCHER with, we explore the implementations provided by code repositories listed as follows:

- TAIL-GNN [13]: `https://github.com/shuaiOKshuai/Tail-GNN`.
- COLBBREW [37]: `https://github.com/amazon-science/gnn-tail-generalization`.
- EERM [26]: `https://github.com/qitianwu/GraphOOD-EERM`.
- GTRANS [8]L `https://github.com/ChandlerBang/GTrans`.
- DGI [24]: `https://github.com/dmlc/dgl/tree/master/examples/pytorch/dgi`.
- GRACE [39]: `https://github.com/dmlc/dgl/tree/master/examples/pytorch/grace`.
- PARETOGNN [9]: `https://github.com/jumxglhf/ParetoGNN`.

We sincerely appreciate the authors of these works for open-sourcing their valuable code and researchers at DGL for providing reliable implementations of these models. For TUNEUP [7], since the authors have not released the code yet, we manually implement it by ourselves, with a similar performance as reported in its original paper.

C Proof to Theorem 1

Here we re-state Theorem 1 before diving into its proof:

Theorem 1. *Assuming the parameters of* GRAPHPATCHER *are initialized from the set* $P_\beta = \{\boldsymbol{\phi} : ||\boldsymbol{\phi} - \mathcal{N}(\mathbf{0}_{|\boldsymbol{\phi}|}; \mathbf{1}_{|\boldsymbol{\phi}|})||_F < \beta\}$ *where* $\beta > 0$*, with probability at least* $1 - \delta$*, for all* $\boldsymbol{\phi} \in P_\beta$*, the error bound (i.e.,* $\mathbb{E}(\mathcal{L}_{patch}) - \mathcal{L}_{patch}$*) is* $\mathcal{O}(\beta\sqrt{\frac{|\boldsymbol{\phi}|}{L}} + \sqrt{\frac{\log(1/\beta)}{L}})$.

Proof. To prove Theorem 1, we need the following lemma, which has been broadly utilized in the literature of generalization error bound [14, 17].

Lemma 1. *Suppose a set* P *of functions is* (B, d)*-Lipschitz parameterized for* $B > 0$ *and* $d \in \mathbb{N}$ *with input from a distribution* D *and output in* $(0, 1)$*. There exist a constant* c *such that for all* $n \in \mathbb{N}$*, for any* $\delta > 0$*, if* S *is obtained by sampling* n *times independently from* D*, with probability at least* $1 - \delta$*, for all* B *and* $f \in P$*, we have:*

$$\mathbb{E}_{d\sim D}[f(d)] - \mathbb{E}_S[f] \leq c \cdot \Big(B\sqrt{\frac{d}{n}} + \sqrt{\frac{\log(1/\delta)}{n}}\Big). \tag{7}$$

In order to prove $\mathbb{E}(\mathcal{L}_{\text{patch}}) - \mathcal{L}_{\text{patch}}$ is $\mathcal{O}(\beta\sqrt{\frac{|\boldsymbol{\phi}|}{L}} + \sqrt{\frac{\log(1/\beta)}{L}})$, we need to show that $\mathcal{L}_{\text{patch}}$ is Lipschitz continuous. $\mathcal{L}_{\text{patch}}$, as discussed in Section 3.2.1, is a regularized cross-entropy formulated as $(\mathbf{y}_1 + \epsilon) \cdot \big(\log(\mathbf{y}_2 + \epsilon) - \log(\mathbf{y}_1 + \epsilon)\big)$. In this work, $\mathbf{y}_1$ and $\mathbf{y}_2$ refers to the prediction distribution (i.e., $0 < \mathbf{y}_1 < 1$) delivered by the GNN we aim at improving. Hence, we need to show that for given a specific $\mathbf{y}_1$, for any two $\mathbf{y}_2^a, \mathbf{y}_2^b \in \{\mathbf{y}_2' : 0 < \mathbf{y}_2' < 1\}$ and $K \in \mathbb{R}^+$, we have

$$\Big|\Big|(\mathbf{y}_1 + \epsilon) \cdot \log(\frac{\mathbf{y}_2^a + \epsilon}{\mathbf{y}_1 + \epsilon}) - (\mathbf{y}_1 + \epsilon) \cdot \log(\frac{\mathbf{y}_2^b + \epsilon}{\mathbf{y}_1 + \epsilon})\Big|\Big|_F \leq K \cdot \Big|\Big|\mathbf{y}_2^a - \mathbf{y}_2^b\Big|\Big|_F \tag{8}$$

$$\Big|\Big|(\mathbf{y}_1 + \epsilon) \cdot \big(\log(\frac{\mathbf{y}_2^a + \epsilon}{\mathbf{y}_1 + \epsilon}) - \log(\frac{\mathbf{y}_2^b + \epsilon}{\mathbf{y}_1 + \epsilon})\big)\Big|\Big|_F \leq K \cdot \Big|\Big|\mathbf{y}_2^a - \mathbf{y}_2^b\Big|\Big|_F \tag{9}$$

$$\Big|\Big|(\mathbf{y}_1 + \epsilon) \cdot \big(\log(\frac{\mathbf{y}_2^a + \epsilon}{\mathbf{y}_2^b + \epsilon})\big)\Big|\Big|_F \leq K \cdot \Big|\Big|\mathbf{y}_2^a - \mathbf{y}_2^b\Big|\Big|_F \tag{10}$$

Given the fact that $\log(\cdot)$ is strictly concave, Equation (10) holds and hence $\mathcal{L}_{\text{patch}}$ is Lipschitz continuous. We can then directly apply Lemma 1 to show that Theorem 1 holds. $\square$