A Details on Experimental Setup of Baselines

Code preparation We downloaded the code of all baselines (vanilla GCN [1], GraphSAGE [2], GAT [3], GIN [4] and MixHop [5]) from corresponding official GitHub repositories. The original version of GIN only worked on graph classification tasks and we adapted it to node classification tasks. The multi-layer perceptrons in GIN have 2 layers.

Hyperparameter selection For baselines containing recommended hyperparameters in the source code, including GAT on `cora`, MixHop, and Vanilla GCN on `cora`, `citeseer`, and `pubmed`, we directly adopted their settings. Meanwhile, we conducted additional tests to adjust their learning rate and model dimension and confirmed that their official settings are almost the best choices. Specifically, GAT had an initial learning rate of 0.005, a hidden state dimension of 8, and 8 attention heads. MixHop's initial learning rates are 1, 0.5, and 0.25 on `cora`, `citeseer`, and `pubmed` respectively. Vanilla GCN had an initial learning rate of 0.01 and a hidden state dimension of 16 [6].

When there is no recommended setting of hyperparameters, or the settings are not affordable on our device, we performed hyperparameter selection on learning rate, dropout rate, and model dimension. We tested the initial learning rates 0.1, 0.01, 0.001, 0.0001 and the dropout rates 0.1, 0.3, 0.5, 0.7, 0.9 for all models. For fair comparison, we constrained the number of layers at 2 and the dimension of hidden states at 128. All methods used a batch size of 512. Specifically, for MixHop, we performed dropout rate selection on input and hidden layers and tested the layer capacities 18, 20, 24, 30. For GIN, we adopted GIN-0 at each layer.

B Details on GRARF Implementation

Constructor We decomposed the actions into two stages. Each stage corresponded to a specific sub-constructor. In both stages, all graph nodes were first encoded with a GraphSAGE layer ($h_u^c = \mathrm{elu}(\boldsymbol{W}_c[x_u \| \frac{1}{n}\sum_{v\in N_a(u)} x_v] + \boldsymbol{b}_c)$. For the state encoder f_s, we used linear transformations of concatenations of central node representations and ARF-averaged node representations, i.e.,

$$f_s(N_a(u)) = \boldsymbol{W}_s[h_u \| \frac{1}{n}\sum_{v\in N_a(u)} h_v] + \boldsymbol{b}_s). \tag{1}$$

For the action encoder f_a , we used the hidden representations of nodes, i.e., $f_a(v) = h_c(v)$. The approximated Q-function in GRARF was parameterized as

$$Q_1(s_t, a_t^1) = \boldsymbol{w}_1^T[f_a(a_t^1) \| f_s(s_t))] + \boldsymbol{b}_1, \tag{2}$$

$$Q_2(s_t, a_t^2; a_t^1) = \boldsymbol{w}_2^T[f_a(a_t^1) \| f_a(a_t^2) \| f_s(s_t))] + \boldsymbol{b}_2, \tag{3}$$

where a_t^1 was the node selected in the first stage. In the training of the constructors, we set the discount rate as $\gamma = 0.9$, and the exploration with linear decay. After 200 steps, the exploration rate decayed to 0.05 and remained unchanged. The size of the memory pool was set as 50000. For each action, the constructor chose a node to add to the ARF among all nodes adjacent to the ARF. Due to memory constraints, we limited the size of nodes adjacent to the ARF to 300 by sampling. This design was for nodes with large numbers of degrees specifically. The hidden representation of states and actions (i.e. outputs of f_s and f_a) were defined to be 128-dimensional.

Hyperparameter selection In all settings, we performed hyperparameter selection on the learning rate, training steps, maximum ARFs size and update frequency of constructor. In the pretraining phase, we trained the evaluator with the same training nodes as in GRARF. The initial learning rate in the pretraining phase was 0.001. In the training phase for GRARF, we performed a parameter

[1] https://github.com/tkipf/gcn

[2] https://gihub.com/williamleif/GraphSAGE

[3] https://gihub.com/PetarV-/GAT

[4] https://gihub.com/weihua916/powerful-gnns

[5] https://github.com/samihaija/mixhop

[6] We tried the hidden state dimension 128 in Vanilla GCN and the hidden state dimension 16 in GAT, while no improvement was observed.

sweep on initial learning rates over $0.01, 0.001, 0.0001$ with step-wise learning rate decay at every 3 steps. The decay γ was set as 0.95. For dataset `cora`, `citeseer`, `pubmed`, we updated the target constructors every 20 steps. For dataset `github`, `ppi`, we updated the target constructors every 30 steps. We performed a parameter sweep on training steps over $400, 500, 600$. The specific batch number in each step was decided by the training size of each dataset. For `cora`, `citeseer` datasets, the maximum ARFs size was set as 2. For `pubmed` dataset, the maximum ARFs size was set as 4, and for `github` and `ppi`, the maximum ARFs size was set as 8. To train the evaluator, we used the *cross-entropy* loss over the softmax output for single-class node classification and the sigmoid output for multi-class node classification; to train the constructor, we used the *mean-square-error* loss.

C Further Analysis

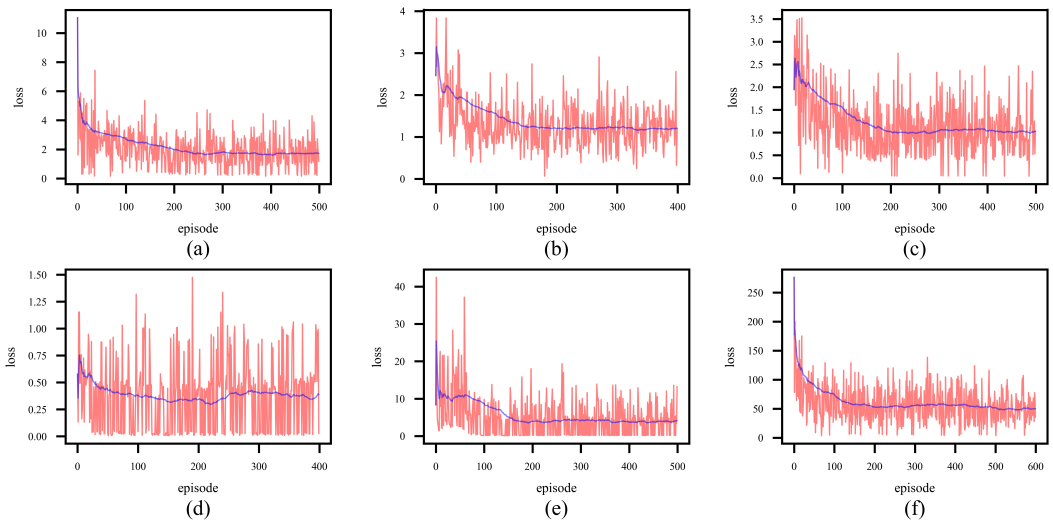

Figure 1: Training loss. (a)-(f) correspond to the training loss curves of `cora`, `citeseer`, `pubmed`, `github` and `ppi` datasets, respectively. The red lines represent step loss and the blue lines represent sliding mean loss (sliding window size is 25).

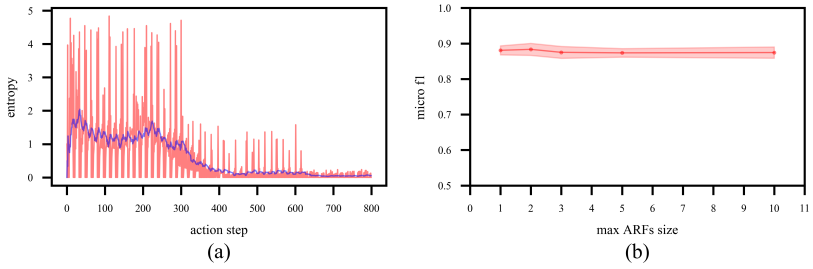

Figure 2: (a) Entropy of each action step. The red line is the entropy of each action step, and the blue line is the sliding mean entropy (sliding window size is 25). (b) *Micro-f1*s of different maximum ARFs sizes and confidence intervals.

Training curves. The training loss curves of GRARF is shown in Figure 1. In our implementation, the step loss would not reach near 0, but converge to a constant value. The strategies given by constructors would also become stable as the number of training steps grown. To illustrate the stability of strategy, we visualized the entropy of q-values in Figure 2 (a) given by constructors. With

the number increasing, the entropy of q-values of each step decreased in fluctuations, and finally reached a constant approaching 0. This shows GRARF learns stable policies.

Effects of Maximum ARFs Sizes To explore the relationship between the performances of GRARF and maximum ARFs sizes, we designed experiments on different maximum ARFs sizes on `cora`. Results are shown in Figure 2 (b). The performance of GRARF remained consistent as the maximum ARF size varies. We believe that this indicates the neighborhood can indeed be depicted with sparse contexts.

Distance Distribution on LDD Experiments To further demonstrate the benefits of using ARFs in LDD, we show the dependency length distributions of nodes having the same degree in Figure 3. In these experiments, maximum ARFs size was set as 5. We selected LDD with $\lambda = 2.0$ and divided central nodes according to their degrees. For low-degree central nodes, ARFs tended to include long-dependency nodes to exploit more information. For high-degree central nodes, ARF nodes with dependency length at 2 and 3 appeared more frequently. Note that in the LDD setup, we assigned a split number k drawn from $Poisson(\lambda)$ on each edge, and then split each edge to a $(k+1)$-hop path (do nothing if $k = 0$) by inserting k noises. Under this setting, informative nodes are pulled away from central nodes, which means GCNs need to "look" further to aggregate informative nodes.

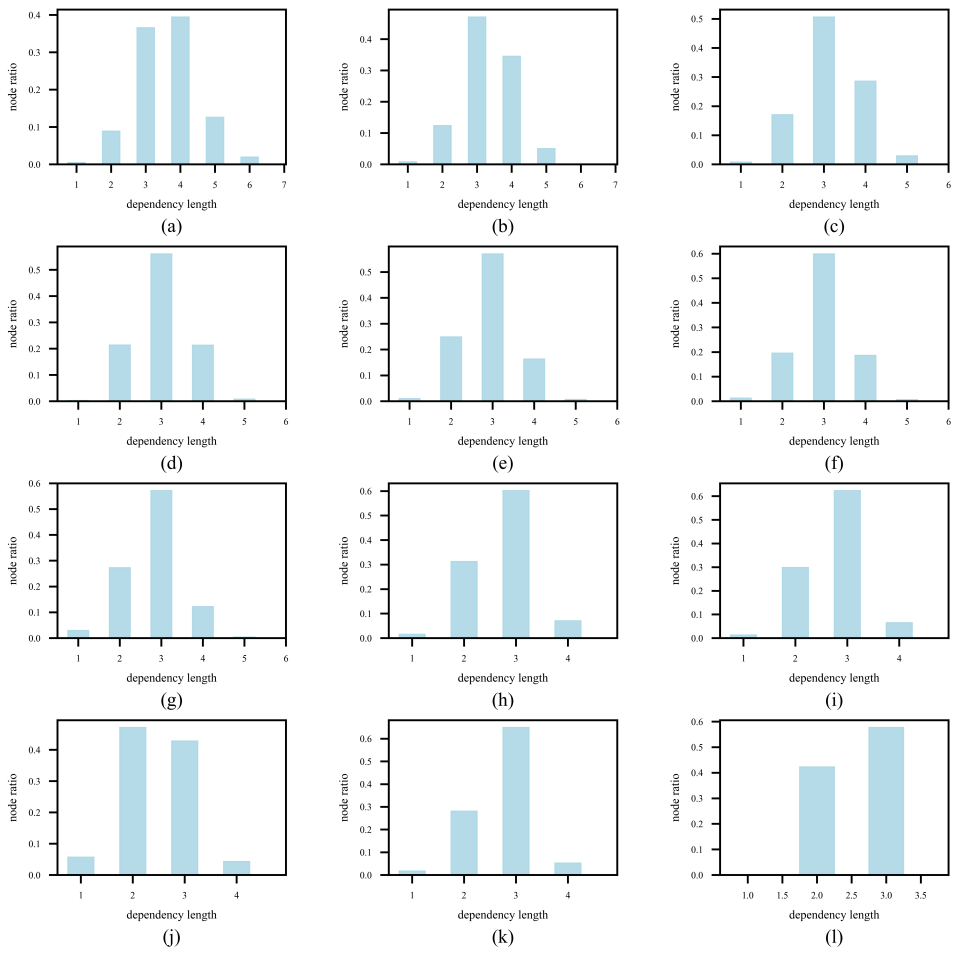

Figure 3: Dependency length of nodes in LDD experiments ($\lambda = 2.0$). (a)-(l) correspond to central nodes of which degrees range from 2 to 13.