Summary of Appendix

We first provide code and discuss reproducibility in Section A. We present dataset details in Section B, method details in Section C, analysis and discussions in Section D, implementation and training details in Section E and extra experiment results in Section F.

A Code and Reproducibility

We provide an anonymous code in the supplementary material. We set the random seed as 2022 for all experiments to enable reproducible results. We provide dataset statistics in Table 5 and details for the proposed graph theory benchmark in Appendix B.2. Details of the hyper-parameters are reported in Table 8. Configuration of all hyper-parameters and the command lines to reproduce the experiments have been included in the code repository.

B Dataset Details

B.1 Dataset Statistics and Metrics

We provide the statistics of all datasets used in our experiments in Table 5 and introduce the evaluation metrics for each dataset.

For Synthetic datasets, we use classification accuracy (ACC) as the evaluation metric. We use Mean Square Error (MSE) as the evaluation metric for all datasets in our Graph Theory Benchmark. For GNN Benchmark, we follow the original work [13] for evaluation, i.e., Mean Absolute Error (MAE) for ZINC and AQSOL, classification accuracy for MNIST and CIFAR10, and balanced classification accuracy for PATTERN and CLUSTER. For OGB Benchmark, we follow the original work [26] and use the ROC-AUC for classification tasks and Root Mean Square Error (RMSE) for regression tasks. For TU datasets, we follow the setting used by [9] and use classification accuracy as the evaluation metric.

B.2 Graph Theory Benchmark

In this section, we provide the details about the tasks and how the graph features and the labels are generated given a base graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$:

- Single source single destination shortest path (SP_{sssd}): a source node $s \in \mathcal{V}$ and a destination node $t \in \mathcal{V}$ are selected uniform randomly. The feature of each node v contains three numbers: (1, whether the node v is s, whether the node v is t). The label of a graph is the length of the shortest path from s to t.
- A maximum connected component of the same color (MCC): each node of the graph is colored with one of three colors. The feature for each node is the one-hot representation of its color. The label of graph is the size of the largest connected component of the same color for each color.
- Graph diameter (Diameter): the label of the graph is the diameter of the graph. The diameter of a graph $\mathcal{G}$ is the maximum of the set of shortest path distances between all pairs of nodes in the graph. The feature of each node is a uniform number 1.
- Single source shortest path (SP_{ss}): a source node s is selected uniformly randomly. The feature of each node contains two numbers: (1, whether the node is s). The label of each node is the length of the shortest path from s to this node.
- Graph eccentricity (ECC): the label of each node v is node's eccentricity in the graph, which is the maximum distance from v to the other nodes. The feature of each node is a uniform number 1.

For each task and graph generation method, We generate the dataset by the following steps:

- Sample N (number of nodes) from $[20, 50]$, totally 300 graphs. These numbers can be configured.

Table 5: The statistics of the datasets used in experiments. Some statistics (like the average number of edges) of the Graph Theory datasets may vary depending on different random graph generation methods. The regression tasks are marked with ✓ in a separate column. The tasks of 4 synthetic datasets are transductive, where the same graph is used for both training and testing. We do not use the node labels as features during the training time. The train-val-test split is over nodes. All other datasets in the table are inductive, where the testing graphs do not occur during training, and the train-val-test split is over graphs.

Collection	Dataset	# Graphs	Avg # Nodes	Avg # Edges	# Node Feat	# Edge Feat	# Classes	Task	Reg.
Synthetic	BaShape	1	700	1761	1	-	4	Trans-Node	
Synthetic	BaCommunity	1	1400	3872	10	-	8	Trans-Node	
Synthetic	TreeCycle	1	871	970	1	-	2	Trans-Node	
Synthetic	TreeGrid	1	1231	1705	1	-	2	Trans-Node	
GraphTheory	SP_{sssd}	300	35.0	-	3	-	-	Graph	✓
GraphTheory	Diameter	300	35.0	-	1	-	-	Graph	✓
GraphTheory	MCC	300	35.0	-	3	-	-	Graph	✓
GraphTheory	SP_{ss}	300	35.0	-	2	-	-	Node	✓
GraphTheory	ECC	300	35.0	-	1	-	-	Node	✓
LRGB	Peptides-func	15535	150.94	307.30	9	3	10	Graph	
LRGB	Peptides-struct	15535	150.94	307.30	9	3	-	Graph	✓
GNNBenchmark	ZINC	12000	23.16	49.83	28	4	2	Graph	✓
GNNBenchmark	AQSOL	9823	17.57	35.76	65	5	2	Graph	✓
GNNBenchmark	MNIST	70000	70.57	564.53	3	1	10	Graph	
GNNBenchmark	CIFAR10	60000	117.63	941.07	5	1	10	Graph	
GNNBenchmark	PATTERN	14000	118.89	6078.57	3	-	2	Node	
GNNBenchmark	CLUSTER	12000	117.20	4301.72	7	-	6	Node	
OGB Graph	molhiv	41127	25.51	80.45	9	3	2	Graph	
OGB Graph	molbace	1513	34.09	107.81	9	3	2	Graph	
OGB Graph	molbbbp	2039	24.06	75.97	9	3	2	Graph	
OGB Graph	molclintox	1477	26.16	81.93	9	3	2	Graph	
OGB Graph	molsider	1427	33.64	104.36	9	3	2	Graph	
OGB Graph	moltox21	7831	18.57	57.16	9	3	2	Graph	
OGB Graph	moltoxcast	8576	18.78	57.30	9	3	2	Graph	
OGB Graph	molesol	1128	13.29	40.64	9	3	-	Graph	✓
OGB Graph	molfreesolv	642	8.7	25.50	9	3	-	Graph	✓
OGB Graph	mollipo	4200	27.04	86.04	9	3	-	Graph	✓
TU	MUTAG	188	17.93	19.79	7	-	3	Graph	
TU	NCI1	4110	29.87	32.30	37	-	2	Graph	
TU	PROTEINS	1113	39.06	72.82	4	-	2	Graph	
TU	D&D	1178	284.32	715.66	89	-	2	Graph	
TU	ENZYMES	600	32.63	62.14	21	-	6	Graph	
TU	IMDB-B	1000	19.77	96.53	10	-	2	Graph	
TU	IMDB-M	1500	13.00	65.94	10	-	3	Graph	
TU	RE-B	2000	429.63	497.75	10	-	2	Graph	
TU	RE-M5K	4999	508.52	594.87	10	-	5	Graph	
TU	RE-M12K	11929	391.41	456.89	10	-	11	Graph	

- Use the graph generation method to generate a graph of N nodes.
- Create graph features and labels according to the task.

We then provide the details about the random graph generation methods we used to create our Graph Theory datasets.

Following [10], we continue to use undirected and unweighted graphs from a wide variety of types. We inherit their 10 random graph generation methods and quote their descriptions here for completeness (the percentage after the name is the approximate proportion of such graphs in the mixture setting).

- **Erdös-Rényi (ER)** (20%) [16]: with a probability of presence for each edge equal to p, where p is independently generated for each graph from $\mathcal{U}[0, 1]$
- **Barabási-Albert (BA)** (20%) [2]: the number of edges for a new node is k, which is taken randomly from $\{1, 2, \ldots, N-1\}$ for each graph
- **Grid** (5%): $m \times k$ 2d grid graph with $N = mk$ and m and k as close as possible
- **Caveman** (5%) [51]: with m cliques of size k, with m and k as close as possible
- **Tree** (15%): generated with a power-law degree distribution with exponent 3
- **Ladder graphs** (5%)
- **Line graphs** (5%)
- **Star graphs** (5%)
- **Caterpillar graphs** (10%): with a backbone of size b (drawn from $\mathcal{U}[1, N)$), and $N - b$ pendent vertices uniformly connected to the backbone
- **Lobster graphs** (10%): with a backbone of size b (drawn from $\mathcal{U}[1, N)$), p (drawn from $\mathcal{U}[1, N-b]$) pendent vertices uniformly connected to the backbone, and additional $N-b-p$ pendent vertices uniformly connected to the previous pendent vertices.

Additional, we add three more graph generation methods:

- **Cycle graphs**
- **Pseudotree graphs**: A tree graph plus an additional edge. The graph is generated by first generating a cycle graph of size $m = \text{sample}(0.3N, 0.6N)$. Then $n - m$ remaining nodes are sampled to m parts, where i-th part represents the size of the tree hanging on the i-th node on the cycle. The trees are randomly generated with the given size.
- **Geographic (Geo) graphs**: geographic threshold graphs, but with added edges via a minimum spanning tree algorithm, to ensure all nodes are connected. This graph generation method is introduced by [6] in their codebase [1]. We use the geographic threshold $\theta = 200$ instead of the default value $\theta = 1000$.

Note that we do not have randomization after the graph generation as in [10]. Therefore, very long diameter is preserved for some type of graphs.

C Method Details

C.1 Cross Update Function

The cross update function $(\mathbf{X}'_j, \hat{\mathbf{X}}'_j, \mathbf{X}'_{j+1}) = \texttt{X-UPD}(j, \mathbf{X}_j, \hat{\mathbf{X}}_j, \mathbf{X}_{j+1})$ perform information exchange in consecutive hierarchies.

The *X-Conv* realization contains the following steps:

1. Merge the node features of $\mathbf{X}_j$ and $\mathbf{X}_{j+1}$ with the inter-graph feature $\hat{\mathbf{X}}_j$, results in $\bar{\mathbf{X}}_j$.
2. Apply GN blocks on inter-graph $\hat{\mathcal{G}}_j$: $\hat{\mathbf{X}}'_j = \text{GN}^{i,j}_{\text{inter}}(\hat{\mathcal{G}}_j, \bar{\mathbf{X}}_j)$.
3. Retrieve $\mathbf{X}'_j$ and $\mathbf{X}'_{j+1}$ from the node features of inter-graph features $\hat{\mathbf{X}}'_j$.

[1] https://github.com/deepmind/graph_nets, the shortest path demo

C.2 S-EdgePool

In this subsection, we introduce the details of S-EdgePool. We first introduce the score generation method, then give details about the SELECT, CONNECT, REDUCE and EXPAND functions, and lastly provide pseudocode of the algorithm.

C.2.1 Edge Score Generation

Both S-EdgePool and EdgePool methods compute a raw edge score $\mathbf{r}_k$ for each edge k using a linear layer:

$$\mathbf{r}_k = \mathbf{W} \cdot (\mathbf{V}_{s_k} || \mathbf{V}_{t_k} || \mathbf{E}_k) + \mathbf{b}$$

where s_k and t_k are the source and target nodes of edge k, $\mathbf{V}$ is node features, $\mathbf{E}$ is edge features, $\mathbf{W}$ and $\mathbf{b}$ are learned parameters. The raw edge scores are further normalized by a local softmax function over all edges of a node:

$$\mathbf{w}_k = \exp(\mathbf{r}_k) / \sum_{k', t_{k'} = t_k} \exp(\mathbf{r}_{k'}),$$

and biased by a constant 0.5 [11].

C.2.2 Select, Connect, Reduce and Expand

SELECT **step.** S-EdgePool shares the same computations as in EdgePool to generate learnable edge scores, as detailed above. Then, we use a clustering procedure to determine the subset of nodes to be reduced.

Let $I_{\mathbf{v}}$ be the identifier of the cluster containing a set of nodes $\mathbf{v}$. Initially, we let $\mathbf{v} = \{v\}$ for every single node v. A contraction of an edge merges a pair of nodes (v, v') connected by this edge (where $v \in \mathbf{v}$, $v' \in \mathbf{v}'$ and $\mathbf{v} \neq \mathbf{v}'$), and thus unifies the cluster identifiers, i.e., $I_{\mathbf{v}} = I_{\mathbf{v}'} = I_{\mathbf{v}_{\text{merge}}}$ and $\mathbf{v}_{\text{merge}} = \mathbf{v} \cup \mathbf{v}'$. That is, once an edge connecting any pair of nodes from two distinct clusters is contracted, we merge the two clusters and unify their identifiers. Edges are visited sequentially by a decreasing order on the edge scores, and contractions are implemented if valid. We set the maximum size of the node clusters to be a parameter τ_c, where $\tau_c = 2$ degenerates to the case of EdgePool [11]. We further introduce the pooling ratio η_v to control the minimal number of remaining clusters after edge contractions to be $N^v * \eta_v$. Contractions that violate the above two constraints are invalid and will be skipped. Both parameters control the number of nodes in the pooled graph. In our implementation, the cluster of nodes is dynamically maintained using the disjoint-set data structure [19].

Then each node cluster i collapses into a new node $\tilde{v}$ of the pooled graph (i.e. $S_{\tilde{v}} = \{v | I_v = i\}$), with inter-graph edges connect the nodes in the cluster to the new node $\tilde{v}$.

CONNECT **step.** The CONNECT function rebuilds the edge set $\tilde{\mathcal{E}}$ between the nodes in $\tilde{\mathcal{V}}$. As aforementioned, we build the pooled graph's nodes according to node clusters. We call this mapping function from node clusters to new nodes as *c2n*. After that, we build the pooled graph's edges following three steps: First, for all edges in the original graph, we find out the corresponding node cluster(s) of its two endpoints (using a disjoint-set's find index operation). Then, we find out the corresponding new nodes by using the mapping function n. Last, we add a new edge between the new nodes.

REDUCE **and** EXPAND **step.** The REDUCE and EXPAND are generalized from the method mentioned in [11]. The REDUCE function computes new node features and edge features. We follow their method to compute new node features by taking the sum of the node features and multiplying it by the edge score. Specifically, we generalize the computation between two nodes to a node cluster. The node clusters are maintained with a disjoint-set data structure and a cluster $S_{\tilde{v}}$ consists of $|S_{\tilde{v}}|$ nodes. We define $\mathcal{E}^{ds}_{\tilde{v}}$ as a set of $|S_{\tilde{v}}| - 1$ edges, where the edges are the selected edges to be contracted in the SELECT step. Then,

$$\begin{aligned} c_{\tilde{v}} &= \frac{1 + \sum_{e_k \in \mathcal{E}^{ds}_{\tilde{v}}} \mathbf{w}_k}{|S_{\tilde{v}}|} \\ \mathbf{V}_{\tilde{v}} &= c_{\tilde{v}} \sum_{v \in S_{\tilde{v}}} \mathbf{V}_v \end{aligned}$$

To integrate the edge features between two node clusters, we first find all the connected edges between the two node clusters (the edges between node clusters are edges that connect two nodes from different node clusters). Then, we use the sum of all the connected edges' features between the two node clusters as the new edge's features.

The `EXPAND` function is also referred as *unpool* operation. It computes node features of the input graph $\mathbf{V}_v$ given the node features of the pooled graph $\mathbf{V}_{\tilde{v}}$ as following:

$$\mathbf{V}_v = \frac{\mathbf{V}_{\tilde{v}}}{c_{\tilde{v}}}$$

C.2.3 Pseudo Code

The pseudo-code includes two parts, where Algorithm 1 describes how to maintain the clusters using a disjoint-set data structure, and Algorithm 2 describes the procedure of S-EdgePool that generates a pooled graph $\tilde{\mathcal{G}}$ with configurable node pooling ratio η_v and maximum of cluster sizes τ_c.

Algorithm 1 Get Cluster Index And Cluster Size of a Node (Using disjoint-set data structure)

```
function InitializeDisjointSet(graph G(V, E))
  for v ∈ V do
    index[v] = v {the identifier of the cluster the node v belongs to}
  end for
end function
function FindIndex(node v)
  if index[v] = v then
    return v
  else
    index[v] ←FindIndex(index[v])
    return index[v]
  end if
end function
function FindIndexAndSize(node v)
  i ← FindIndex(v)
  s ← size[i]
  return i, s
end function
function MERGE(cluster index x, cluster index y)
  size[y] ← size[x] + size[y]
  index[x] ← index[y]
end function
```

C.3 GFuN

We first realize the ϕ_e, ϕ_v, ϕ_u functions in the full GN block (Sec 2.1 and [6]) as neural networks:

$$\begin{aligned}
\mathbf{E}'_k &= \mathrm{NN}_e(\mathbf{E}_k, \mathbf{V}_{s_k}, \mathbf{V}_{t_k}, \mathbf{u}), && (2)\\
\mathbf{V}'_i &= \mathrm{NN}_v(\bar{\mathbf{E}}'_i, \mathbf{V}_i, \mathbf{u}), && (3)\\
\mathbf{u}' &= \mathrm{NN}_u(\bar{\mathbf{E}}', \bar{\mathbf{V}}', \mathbf{u}), && (4)
\end{aligned}$$

respectively, where

$$\begin{aligned}
\bar{\mathbf{E}}'_i &= \rho^{e\to v}(\{\mathbf{E}'_k\}_{k\in[1\ldots N^e], t_k=i}), && (5)\\
\bar{\mathbf{E}}' &= \rho^{e\to u}(\mathbf{E}'), && (6)\\
\bar{\mathbf{V}}' &= \rho^{v\to u}(\mathbf{V}'). && (7)
\end{aligned}$$

We further decompose the neural networks according to the features in the function:

$$\begin{aligned}
\mathrm{NN}_e(\mathbf{E}_k, \mathbf{V}_{s_k}, \mathbf{V}_{t_k}, \mathbf{u}) &= \mathrm{NN}_{e\leftarrow e}(\mathbf{E}_k) + \mathrm{NN}_{e\leftarrow v_s}(\mathbf{V}_{s_k}) + \mathrm{NN}_{e\leftarrow v_t}(\mathbf{V}_{t_k}) + \mathrm{NN}_{e\leftarrow u}(\mathbf{u}), && (8)\\
\mathrm{NN}_v(\bar{\mathbf{E}}'_i, \mathbf{V}_i, \mathbf{u}) &= \mathrm{NN}_{v\leftarrow e}(\bar{\mathbf{E}}'_i) + \mathrm{NN}_{v\leftarrow v}(\mathbf{V}_i) + \mathrm{NN}_{v\leftarrow u}(\mathbf{u}), && (9)\\
\mathrm{NN}_u(\bar{\mathbf{E}}', \bar{\mathbf{V}}', \mathbf{u}) &= \mathrm{NN}_{u\leftarrow e}(\bar{\mathbf{E}}') + \mathrm{NN}_{u\leftarrow v}(\bar{\mathbf{V}}') + \mathrm{NN}_{u\leftarrow u}(\mathbf{u}) && (10)
\end{aligned}$$

Algorithm 2 Strided EdgePool

input graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, edge scores $\mathbf{w}$, node pooling ratio η_v, maximum cluster sizes τ_c.
output pooled graph $\tilde{\mathcal{G}} = (\tilde{\mathcal{V}}, \tilde{\mathcal{E}})$ and inter graph $\hat{\mathcal{G}} = (\hat{\mathcal{V}}, \hat{\mathcal{E}})$

```
InitializeDisjointSet(G)
remains ← N^v {N^v is the number of nodes in graph G}
Ē ← Sort the edges E according to the edge scores w decreasingly.
for e ∈ Ē do
    x, y ← the two endpoints of the edge e
    rx, sx ← FindIndexAndSize(x)
    ry, sy ← FindIndexAndSize(y)
    if rx ≠ ry and (sx + sy ≤ τ_c) then
        Merge(x, y)
        remains ← remains − 1
        if remains ≤ N^v * η_v then
            break
        end if
    end if
end for
Ṽ, Ẽ, V̂, Ê ← {}, {}, {}, {}
create empty mapping c2n from cluster index to nodes
for v ∈ V do
    if FindIndex(v) = v then
        create new node ṽ
        c2n[v] = ṽ
        Ṽ ← Ṽ ∪ {ṽ}
    end if
end for
for e ∈ E do
    x, y ← the two endpoints of the edge e
    x̃ ← c2n[FindIndex(x)]
    ỹ ← c2n[FindIndex(y)]
    Ẽ ← Ẽ ∪ {(x̃, ỹ)}
end for
for v ∈ V do
    ṽ ← c2n[FindIndex(v)]
    Ê ← Ê ∪ {(v, ṽ)}
end for
V̂ ← V ∪ Ṽ
```

However, such GN block uses 10 times the number of parameters as the standard GCN [31] layer when the node, edge and global embedding dimensions are all equivalent. In practice, we disable all computations related to global features $\mathbf{u}$, as well as the neural networks $\mathrm{NN}_{e\leftarrow e}$ and $\mathrm{NN}_{e\leftarrow v_t}$. We also set $\mathrm{NN}_{v\leftarrow e}$ to be Identity.

In practice, we use the summation function as the aggregator function $\rho^{e\rightarrow v}$ by default. But other choices like MEAN, MAX, gated summation, attention or their combinations can also be used.

Overall, we call such GN block as graph full network (GFuN).

C.4 Encorder and Decoder

Encoder. For input embedding, we use the Linear layer or Embedding layer to embed input features. For example, we follow [13] and use the Linear layer on MNIST and CIFAR10 datasets, and use the Embedding layer on ZINC and AQSOL datasets. For the molecular graph in OGB, we use the same embedding method as in the original work [26]. Besides, we can adopt positional encoding methods like Laplacian [13] and Random Walk [14] to further embed global and local graph structure information. The embedding of positional encoding can be combined into (like concatenation, addition, etc.) input features and form new embeddings.

Decoder. We can freely choose from the multi-scale features computed during the *process* stage as inputs to the decoder module. Empirically, we use the features on the original graph for prediction in all experiments. For node-level tasks, we apply a last GNN layer on the original graph to get logits for every node. For graph-level tasks, we first use global pooling functions to aggregate features. We can use common global pooling methods like SUM, MEAN, MAX, or their combination. After the global pool, we use MLP layer(s) to generate the prediction.

C.5 Architecture Variants

We can replace some GN blocks within `Mee` layers as an Identity block to reduce the time complexity. We call the height j is reserved if the intra GN block of height j is not replaced by an Identity block. We prefer to reserve an interval of consecutive heights for the `Mee` layers. (The inter GN blocks between these heights remain unchanged while others are replaced as identities) By varying the heights reserved in each `Mee` layers, we can create a large number of variants of `MeGraph` model including U-Shaped, Bridge-Shaped and Staircase-Shaped.

U-Shaped. This variant is similar to Graph U-Net [20]. In this U-Shaped variant, the relationship between the number of layers n and height h is $n = 2h + 1$, and there is only one GN block in each layer. We keep the GN block at height $j = i$ for each layer i at the first half layers and keep the GN block at height $j = n - i + 1$ for each layer i at the later half layers. In the middle layer, only the last height $j = h = (n - 1)/2$ has a GN block.

Bridge-Shaped. In this variant, all GN blocks are combined like an arch bridge. Describe in detail, in the first and last layers, there are GN blocks in each height. In other layers, there are GN blocks at a height of 1 to j (where $1 < j < h$).

Staircase-Shaped. There are four forms in this variant, and the number of layers n is equal to the height h in all forms. The first form is like the 'downward' staircase. In each layer i of this form, there are GN blocks at the height of j to h (where $j = i$). The second form is the inverted first form. In each layer i of this second form, there are GN blocks at height of 1 to $h - i + 1$ (where $j = i$). The last two forms are the mirror of the first and second forms.

D Theoretical Discussions

D.1 Smaller Number of Aggregation Steps for Capturing Long-Range Interactions

We rephrase the analysis provided in [43] as following:

We analyze the number of aggregation steps required to capture long-range interactions between nodes in the original graph while assuming the node representation capacity is large enough.

Standard message-passing GNNs require n aggregation steps to capture long-range interactions of n hops away, therefore requiring a stack of n layers, which could be expensive when n is large.

We also assume the height h of the hierarchy is large enough so that all nodes of the original graph are pooled into a single node. In that case, the information aggregation along the hierarchy captures all pairs of LRIs into the embedding of the single node. Which means the number of aggregation steps of `MeGraph` is h. When we adopt a pooling method that coarsens the graph at least half, h is at most $O(\log(|V|))$ where $|V|$ is the number of nodes of the input graph. Therefore, the height h is significantly smaller than the diameter of the graph (which could be $O(|V|)$) in most cases.

D.2 `MeGraph` can degenerate to standard GNNs

`MeGraph` can learn a gating function (within the X-UPD function) that only reserves the features of the same scale while performing cross-scale information exchanging. In that case, there will be no information exchange across multi-scale graphs, and features other than those in the original scale will not be aggregated. We provide a proof sketch below.

Proof: The cross update function is $(X'_j, \hat{X}'_j, X'_{j+1}) = \texttt{X-UPD}(j, X_j, \hat{X}_j, X_{j+1})$. There is a residual function applied here, and we assume it is implemented as a gated residual: $X''_j = \sigma(\alpha)X_j + \sigma(\beta)X'_j$, where σ is the sigmoid function and α, β are learnable parameters. Theoremetically, it is possible that $\sigma(\alpha) = 1$ and $\sigma(\beta) = 0$ after training. In that case, $X''_j = X_j$, which means X_j is not changed over

Table 6: Running time (s) for one epoch on the GNN benchmark. See Sec. E for more implementation details.

	ZINC	AQSOL	CIFAR10	MNIST	PATTERN	CLUSTER
Megraph ($h = 5$)	25.69	20.22	336.63	307.23	101.52	69.65
Megraph ($h = 1$)	2.41	1.67	51.74	38.60	9.21	6.52

Table 7: Running time (s) for one epoch on the OGBG datasets. See Sec. E for more implementation details.

	molhiv	molbace	molbbbp	molclintox	molsider
Megraph ($h = 5$)	393.42	14.70	20.36	14.03	14.12
Megraph ($h = 1$)	22.50	1.43	1.58	1.26	1.41
	moltox21	moltoxcast	molesol	molfreesolv	mollipo
Megraph ($h = 5$)	70.15	78.77	11.68	6.24	44.77
Megraph ($h = 1$)	5.27	8.17	0.76	0.41	2.77

steps 2 and 3 of the Mee layer. Therefore, $X_0^i = GN_{intra}^{i,0}(\mathcal{G}, X_0^{i-1})$, this is equivalent to a simple GNN layer that $X^i = GNN_i(\mathcal{G}, X^{i-1})$ as X_0^i is the features of the original graph and GN_{intra} is a GNN layer. Therefore, `MeGraph` degenerates to standard message-passing GNNs in this case. ■

E Implementation and Training Details

We use PyTorch [41] and Deep Graph Library (DGL) [50] to implement our method.

We implement S-EdgePool using DGL, extending from the original implementation of EdgePool in the Pytorch Geometric library (PYG) [17]. We did Constant optimization over the implementation to speed up the training and inference of the pooling. We further use Taichi-Lang [27] to speed up the dynamic node clustering process of S-EdgePool. The practical running time of `MeGraph` model with height $h > 1$ after optimization is about $2h$ times as the $h = 1$ baseline. This is still slower than the theoretical computational complexity due to the constant in the implementation and the difficulty of paralleling the sequential visitation of edges (according to their scores) in the EdgePool and S-EdgePool. This process could be further sped up by implementing the operations with the CUDA library. We provide the practical running time for $h > 1$ and $h = 1$ in GNN benchmark and OGB-G datasets in Tables 6 and 7.

We run all our experiments on V100 GPUs and M40 GPUs. For training the neural networks, we use Adam [30] as the optimizer. We report the hyper-parameters of the `Megraph` in Table 8.

For models using GFuN layer as the core GN block, we find it benefits from using layer norms [4]. However, for models using GCN layer as the core GN block, we find it performs best when using batch norms [28].

The code along with the configuration of hyper-parameters to reproduce our experiments is provided in the Supplementary Material and will be made public.

F Additional Experiment Results

F.1 Experimental Protocol

We evaluate `MeGraph` on public real-world graph benchmarks. To fairly compare `MeGraph` with the baselines, we use the following experimental protocols. We first report the public baseline results and our reproduced standard GCN's results. We then replace GCN layers with GFuN layers (which is equivalent to `MeGraph` ($h = 1$)) to serve as another baseline. We tune the hyper-parameters (such as learning rate, dropout rate and the readout global pooling method, etc.) of `MeGraph` ($h = 1$) and choose the best configurations. We then run other diversely configured `MeGraph` candidates by tuning

Table 8: Hyper-parameters of the standard version of `MeGraph` for each dataset. It is worth noting that the total number of GNN layers is equals to one plus the number of `Mee` layers as $n + 1$.

Hyper-parameters	Synthetic Datasets	Graph Theory Benchmark	LRGB Benchmark	GNN Benchmark	OGB Benchmark	TU Datasets
Repeated Runs	10	5	4	4	5	1 for each fold
Epochs per run	200 for BA* 500 for Tree*	300 (200 for MCC)	200	200 (100 for MNIST, CIFRA10)	100	100 (200 for ENZYMES)
Learning rate	0.002	0.002 (0.005 for MCC)	0.001	0.001	0.001	0.002
Weight decay	0.0005	0.0005	0	0	0.0005	0.0005
Node hidden dim	64	128	160	144	300	128
Edge hidden dim (for GFuN)	64	128	160	144	300	128
Num `Mee` layers n	-	-	4	3	4	2
Height h	-	-	9	5	5	3 or 5
Batch size	32	32	128	128	32	128
Input embedding	False	True	True	True	True	True
Global pooling	Mean	Mean Max	Mean Max Sum	Mean	Mean	Mean Max Sum
Dataset split (train:val:test)	8:1:1	8:1:1	Original split	Original split	Original split	10-fold cross validation

other hyper-parameters that only matters for $h > 1$, and these hyper-parameters are referred to as the `MeGraph` hyper-parameters. Detailed configurations are put in Table 8 in App. E.

F.2 Other Real-Wrold Datasets

TU dataset consists of over 120 datasets of varying sizes from a wide range of applications. We choose 10 datasets, 5 of which are molecule datasets (MUTAG, NCI1, PROTEINS, D&D and ENZYMES) and the other 5 are social networks (IMDB-B, IMDB-M, REDDIT-BINARY, REDDIT-MULTI-5K and REDDIT-MULTI-12K). They are all graph classification tasks. For more details of each dataset, please refer to the original work [39].

Our `Megraph` uses the same network structure and hyper-parameters for the same type of dataset. As shown in Table 9, our `Megraph` achieves about 1% absolute gain than the $h = 1$ Baselines.

F.3 GFuN

We show our GFuN results on real-world datasets compared to our reproduced GCN in Table 10, 11 and 12. Both GCN and GFuN have the same hyper-parameters except the batch norm for GCN and layer norm for GFuN as stated in Appendix E.

F.4 Synthetic Datasets

Figure 5 shows the influence of the height h and the number of `Mee` layers n for `MeGraph` model on the BAShape and BACommunity datasets. The trend on these easier datasets is similar to that on TreeCycle and TreeGrid but less significant.

Table 9: Results on Tu Dataset. † means the results taken from [9] (*: The result of GCN on ENZYMES is 100 epoch).

Model	MUTAG ↑	NCI1 ↑	PROTEINS ↑	D&D ↑	ENZYMES ↑	Average
GCN†	87.20 ±5.11	83.65 ±1.69	75.65 ±3.24	79.12 ±3.07	66.50 ±6.91*	78.42
GIN†	89.40 ±5.60	82.70 ±1.70	76.20 ±2.80	-	-	-
GCN	92.46 ±6.55	82.55 ±0.99	77.82 ±4.52	80.56 ±2.40	74.17 ±5.59	81.51
MeGraph (h=1)	93.01 ±6.83	82.53 ±1.89	81.32 ±4.08	81.32 ±3.17	74.83 ±3.20	82.60
MeGraph	93.07 ±6.71	83.99 ±0.98	81.41 ±3.10	81.24 ±2.39	**75.17** ±4.86	82.98
MeGraph$_{\text{best}}$	**94.12** ±5.02	**84.40** ±1.11	**81.68** ±3.40	**82.00** ±2.86	**75.17** ±4.86	**83.47**

Model	IMDB-B ↑	IMDB-M ↑	RE-B ↑	RE-M5K ↑	RE-M12K ↑	Average
GCN	**76.00** ±3.44	50.33 ±1.89	91.15 ±1.63	56.47 ±1.54	48.71 ±0.88	64.53
MeGraph (h=1)	68.60 ±3.53	51.33 ±2.23	93.10 ±1.16	57.47 ±2.31	51.56 ±1.06	64.41
MeGraph	72.40 ±2.80	51.27 ±2.71	**93.75** ±1.25	57.69 ±2.22	52.03 ±0.86	65.43
MeGraph$_{\text{best}}$	74.30 ±2.97	**52.00** ±2.49	**93.75** ±1.25	**58.45** ±2.22	**52.13** ±1.01	**66.13**

Table 10: Comparison between GCN and GFuN on GNN benchmark.

Model	ZINC ↓	AQSOL ↓	MNIST ↑	CIFAR10 ↑	PATTERN ↑	CLUSTER ↑
GCN	0.426 ±0.015	1.397 ±0.029	90.140 ±0.140	51.050 ±0.390	84.672 ±0.054	47.541 ±0.940
GFuN	0.364 ±0.003	1.386 ±0.024	95.560 ±0.190	61.060 ±0.500	84.845 ±0.021	58.178 ±0.079
MeGraph	0.260 ±0.005	1.002 ±0.021	97.860 ±0.098	69.925 ±0.631	86.507 ±0.067	68.603 ±0.101

F.5 Varying GN block

We vary the aggregation function of the GN block as attention (w/ ATT) and gated function (w/ GATE). We observe similar results as in Sec. 4.2 and 4.3, verifying the robustness of MeGraph over different GN blocks. Results are shown in Tables 15, 16 and 17.

F.6 Graph Theory Dataset

We provide a list of tables (from Table 18 to 28) showing the individual results of Table 1 for each possible graph generation method. Each table contains a list of variants of models and 5 tasks. Some graph generation methods and task combinations are trivial so we filter them out.

Table 11: Comparison between GCN and GFuN on OGB-G.

Model	molhiv ↑	molbace ↑	molbbbp ↑	molclintox ↑	molsider ↑
GCN	75.40 ±1.29	76.01 ±3.31	67.35 ±0.96	89.62 ±2.27	58.08 ±0.78
GFuN	78.54 ±1.14	71.77 ±2.15	67.56 ±1.11	89.77 ±3.48	58.28 ±0.51
MeGraph	77.20 ±0.88	78.52 ±2.51	69.57 ±2.33	92.04 ±2.19	59.01 ±1.45

Model	moltox21 ↑	moltoxcast ↑	molesol ↓	molfreesolv ↓	mollipo ↓
GCN	75.11 ±0.41	64.13 ±0.52	1.141 ±0.02	2.407 ±0.15	0.788 ±0.01
GFuN	75.89 ±0.45	64.49 ±0.46	1.079 ±0.02	2.017 ±0.08	0.768 ±0.00
MeGraph	78.11 ±0.47	67.67 ±0.53	0.886 ±0.02	1.876 ±0.05	0.726 ±0.00

Table 12: Comparison between GCN and GFuN on Tu Dataset.

Model	MUTAG ↑	NCI1 ↑	PROTEINS ↑	D&D ↑	ENZYMES ↑	Average
GCN	92.46 ±6.55	82.55 ±0.99	77.82 ±4.52	80.56 ±2.40	74.17 ±5.59	81.51
GFuN	93.01 ±7.96	82.80 ±1.30	80.60 ±3.83	82.43 ±2.60	73.00 ±5.31	82.37

Model	IMDB-B ↑	IMDB-M ↑	RE-B ↑	RE-M5K ↑	RE-M12K ↑	Average
GCN	76.00 ±3.44	50.33 ±1.89	91.15 ±1.63	56.47 ±1.54	48.71 ±0.88	64.53
GFuN	68.90 ±3.42	51.27 ±3.22	92.25 ±1.12	57.53 ±1.31	51.54 ±1.19	64.30

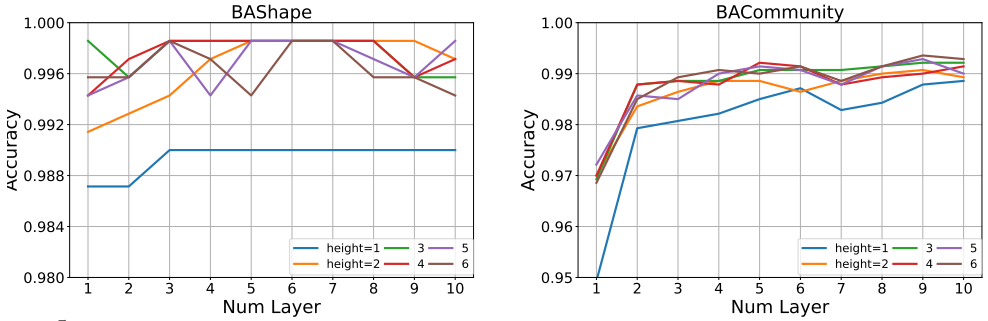

Figure 5: Node Classification accuracy for `MeGraph` model on BAShape (left) and BACommunity (right) datasets, varying the height h and the number of `Mee` layers n. A clear gap can be observed between heights 1 and ≥ 2. The concrete number of accuracy can be found in Table 14.

Table 13: Node Classification accuracy for `MeGraph` model on TreeCycle (above) and TreeGrid (below).

layer \ height	1	2	3	4	5	6
1	61.48 ±6.04	76.59 ±4.41	91.48 ±2.70	98.52 ±1.69	97.95 ±2.32	98.52 ±1.35
2	67.27 ±6.91	81.59 ±4.03	97.39 ±1.25	98.98 ±0.94	98.75 ±1.29	98.86 ±1.14
3	74.43 ±3.60	90.80 ±2.61	98.64 ±1.11	99.09 ±1.11	98.75 ±1.56	99.09 ±0.85
4	79.55 ±4.34	93.41 ±2.82	99.20 ±0.73	99.20 ±0.89	99.66 ±0.73	99.20 ±1.69
5	82.73 ±4.06	93.41 ±1.89	99.43 ±1.05	99.20 ±1.35	99.32 ±1.16	99.32 ±0.56
6	83.18 ±3.51	94.09 ±2.02	99.43 ±0.76	99.09 ±0.85	99.20 ±1.69	99.20 ±0.89
7	84.43 ±3.74	94.43 ±2.24	99.89 ±0.34	99.20 ±1.02	99.20 ±0.89	99.66 ±0.73
8	84.20 ±3.82	94.20 ±2.00	98.98 ±1.19	99.32 ±0.75	99.66 ±0.52	99.20 ±0.73
9	84.43 ±3.87	94.20 ±2.06	99.77 ±0.45	99.20 ±1.02	98.98 ±1.07	99.32 ±0.75
10	84.77 ±3.98	94.43 ±2.18	99.32 ±0.75	98.86 ±1.14	99.09 ±1.67	99.66 ±0.52

layer \ height	1	2	3	4	5	6
1	79.11 ±3.07	91.13 ±2.01	96.85 ±1.11	97.18 ±1.31	97.10 ±1.45	97.42 ±1.34
2	89.68 ±1.76	93.55 ±1.53	98.31 ±0.76	97.82 ±1.14	97.42 ±1.24	97.98 ±0.74
3	90.81 ±1.36	96.13 ±1.48	97.66 ±1.22	98.23 ±0.94	98.87 ±0.82	98.39 ±0.62
4	91.53 ±1.04	96.69 ±1.05	98.06 ±1.03	98.55 ±1.01	98.63 ±1.08	97.98 ±1.15
5	93.95 ±1.58	96.13 ±1.76	98.47 ±1.17	98.47 ±0.92	98.31 ±0.84	97.90 ±0.65
6	94.35 ±1.25	96.69 ±1.46	98.06 ±1.03	98.31 ±1.05	98.15 ±1.20	98.39 ±1.20
7	94.76 ±1.10	97.02 ±1.44	98.47 ±0.84	98.47 ±1.05	98.71 ±0.74	98.87 ±0.90
8	95.08 ±0.76	97.02 ±1.20	98.55 ±1.24	98.87 ±0.82	98.47 ±0.92	98.71 ±1.15
9	94.68 ±1.09	96.94 ±1.19	98.47 ±0.43	98.15 ±1.20	98.15 ±0.89	98.39 ±1.08
10	94.84 ±1.21	96.77 ±1.20	98.47 ±0.92	97.98 ±1.50	98.15 ±1.02	98.23 ±1.19

Table 14: Node Classification accuracy for `MeGraph` model on BAShape (above) and BACommunity (below).

layer \ height	1	2	3	4	5	6
1	98.71 ±1.00	99.14 ±1.14	99.86 ±0.43	99.43 ±0.70	99.43 ±0.95	99.57 ±0.91
2	98.71 ±1.00	99.29 ±0.96	99.57 ±0.91	99.71 ±0.57	99.57 ±0.91	99.57 ±0.91
3	99.00 ±0.91	99.43 ±0.95	99.86 ±0.43	99.86 ±0.43	99.86 ±0.43	99.86 ±0.43
4	99.00 ±0.91	99.71 ±0.57	99.86 ±0.43	99.86 ±0.43	99.43 ±0.95	99.71 ±0.57
5	99.00 ±0.91	99.86 ±0.43	99.86 ±0.43	99.86 ±0.43	99.86 ±0.43	99.43 ±0.95
6	99.00 ±0.91	99.86 ±0.43	99.86 ±0.43	99.86 ±0.43	99.86 ±0.43	99.86 ±0.43
7	99.00 ±0.91	99.86 ±0.43	99.86 ±0.43	99.86 ±0.43	99.86 ±0.43	99.86 ±0.43
8	99.00 ±0.91	99.86 ±0.43	99.86 ±0.43	99.86 ±0.43	99.71 ±0.57	99.57 ±0.65
9	99.00 ±0.91	99.86 ±0.43	99.57 ±0.91	99.57 ±0.91	99.57 ±0.91	99.57 ±0.91
10	99.00 ±0.91	99.71 ±0.57	99.57 ±0.91	99.71 ±0.57	99.86 ±0.43	99.43 ±0.95

layer \ height	1	2	3	4	5	6
1	94.93 ±1.30	97.00 ±1.80	96.93 ±1.60	97.00 ±1.88	97.21 ±1.70	96.86 ±1.67
2	97.93 ±0.87	98.36 ±0.72	98.79 ±0.46	98.79 ±0.46	98.57 ±0.55	98.50 ±1.03
3	98.07 ±0.91	98.64 ±0.87	98.86 ±0.91	98.86 ±0.80	98.50 ±0.98	98.93 ±0.80
4	98.21 ±0.97	98.86 ±0.65	98.86 ±0.80	98.79 ±0.64	99.00 ±0.73	99.07 ±0.64
5	98.50 ±0.87	98.86 ±0.91	99.07 ±0.64	99.21 ±0.67	99.14 ±0.70	99.00 ±0.73
6	98.71 ±0.83	98.64 ±0.87	99.07 ±0.64	99.14 ±0.70	99.07 ±0.85	99.14 ±0.70
7	98.29 ±0.91	98.86 ±0.65	99.07 ±0.56	98.79 ±0.56	98.79 ±0.72	98.86 ±0.57
8	98.43 ±0.77	99.00 ±0.47	99.14 ±0.53	98.93 ±0.58	99.14 ±0.29	99.14 ±0.43
9	98.79 ±0.79	99.07 ±0.56	99.21 ±0.50	99.00 ±0.73	99.29 ±0.45	99.36 ±0.50
10	98.86 ±0.73	98.93 ±0.80	99.21 ±0.87	99.14 ±0.70	99.00 ±0.73	99.29 ±0.64

Table 15: Comparison results of `MeGraph` with ATT and GATE on Graph Theory Benchmark.

Category	Model	SP_{sssd}	MCC	Diameter	SP_{ss}	ECC
`MeGraph` w. ATT	$h = 1$	2.990 ±3.411	3.346 ±3.228	44.41 ±36.33	16.39 ±13.57	29.04 ±27.59
	$h = 5$	0.594 ±0.903	1.706 ±1.409	3.256 ±2.956	1.018 ±1.071	14.80 ±17.09
	$h = 5, \eta_v = 0.3, \tau_c = 4$	0.749 ±1.131	1.128 ±0.794	4.430 ±4.329	0.640 ±0.833	5.649 ±4.496
`MeGraph` w. GATE	$h = 1$	4.144 ±4.181	0.908 ±0.934	6.343 ±7.152	13.94 ±12.78	19.73 ±19.47
	$h = 5$	0.809 ±0.993	0.660 ±0.601	2.506 ±2.639	0.669 ±0.546	7.508 ±7.558
	$h = 5, \eta_v = 0.3, \tau_c = 4$	0.602 ±0.622	0.599 ±0.520	0.544 ±0.490	0.342 ±0.193	0.859 ±0.712

Table 16: Comparison results of `MeGraph` with ATT and GATE on GNN Benchmark.

	ZINC	AQSOL	CIFAR10	MNIST	PATTERN	CLUSTER
`MeGraph` w. ATT ($h = 1$)	0.4258 ±0.0054	1.1421 ±0.0270	69.890 ±0.209	97.570 ±0.168	78.232 ±0.827	59.497 ±0.207
`MeGraph` w. ATT ($h = 5$)	0.3637 ±0.0116	1.0767 ±0.0105	69.925 ±0.631	97.860 ±0.098	83.798 ±0.885	68.930 ±68.76
`MeGraph` w. GATE ($h = 1$)	0.3336 ±0.0036	1.0766 ±1.0556	64.200 ±0.586	96.812 ±0.205	85.391 ±0.029	59.321 ±0.290
`MeGraph` w. GATE ($h = 5$)	0.2897 ±0.0291	1.0240 ±0.0098	64.935 ±0.829	97.290 ±0.140	86.611 ±0.041	67.122 ±3.323

Table 17: Comparison results of `MeGraph` with ATT and GATE on OGB-G.

Model	molhiv ↑	molbace ↑	molbbbp ↑	molclintox ↑	molsider ↑
`MeGraph` w. ATT ($h = 1$)	77.33 ±0.78	74.83 ±4.87	64.74 ±1.14	86.34 ±1.04	58.12 ±0.53
`MeGraph` w. ATT ($h = 5$)	77.15 ±1.37	76.13 ±3.85	68.68 ±2.07	87.17 ±0.76	58.03 ±1.58
`MeGraph` w. GATE ($h = 1$)	76.35 ±0.70	76.36 ±1.55	65.97 ±1.98	87.12 ±0.74	58.11 ±1.29
`MeGraph` w. GATE ($h = 5$)	78.14 ±0.91	78.90 ±1.29	66.53 ±0.74	89.02 ±2.56	59.58 ±1.88

Model	moltox21 ↑	moltoxcast ↑	molesol ↓	molfreesolv ↓	mollipo ↓
`MeGraph` w. ATT ($h = 1$)	75.88 ±0.64	64.65 ±0.59	1.091 ±0.030	2.318 ±0.089	0.790 ±0.012
`MeGraph` w. ATT ($h = 5$)	76.71 ±0.98	66.98 ±0.70	1.007 ±0.617	2.065 ±0.151	0.736 ±0.023
`MeGraph` w. GATE ($h = 1$)	75.30 ±0.43	64.34 ±0.62	1.064 ±0.015	2.191 ±0.068	0.766 ±0.008
`MeGraph` w. GATE ($h = 5$)	76.91 ±0.25	66.78 ±0.13	1.003 ±0.086	2.048 ±0.199	0.700 ±0.014

Table 18: Graph Theory Benchmark results on Grid graphs, all results are obtained using our codebase.

Category	Model	SP_{sssd}	MCC	Diameter	SP_{ss}	ECC
Baselines (h=1)	n=1	6.60 ±0.541	1.50 ±0.050	22.49 ±1.36	26.74 ±0.347	20.99 ±0.232
	n=5	4.18 ±0.737	1.29 ±0.124	5.04 ±1.26	15.54 ±0.155	20.32 ±0.326
	n=10	3.70 ±0.422	1.33 ±0.100	0.737 ±0.116	7.24 ±0.243	20.32 ±0.422
MeGraph(h=5) EdgePool(τ_c=2)	n=1	1.19 ±0.486	1.24 ±0.154	6.78 ±1.95	5.34 ±0.265	18.00 ±0.910
	n=5	0.738 ±0.322	1.11 ±0.043	0.616 ±0.310	0.617 ±0.099	13.3 ±3.31
MeGraph S-EdgePool Variants (h=5, n=5)	τ_c=3	0.361 ±0.182	1.24 ±0.113	0.382 ±0.120	0.442 ±0.130	0.918 ±0.220
	η_v=0.3	4.77 ±2.50	1.33 ±0.161	0.349 ±0.074	5.40 ±0.954	3.59 ±0.354
	η_v=0.3, τ_c=4	0.745 ±0.316	1.35 ±0.168	0.385 ±0.180	0.552 ±0.113	0.622 ±0.100
	η_v=0.5, τ_c=4	1.61 ±0.394	1.28 ±0.138	0.458 ±0.220	1.71 ±0.535	1.48 ±0.283
	η_v=0.3, τ_c=4 (X-Pool)	1.03 ±0.365	1.50 ±0.142	0.626 ±0.216	1.70 ±0.185	3.44 ±0.991
	η_v=0.3, τ_c=4 (w/o pw)	0.616 ±0.194	1.66 ±0.083	0.361 ±0.147	0.678 ±0.139	1.70 ±0.485
Graph-UNets	h=5,n=9,η_v=0.3,τ_c=4	0.969 ±0.643	2.18 ±0.381	0.111 ±0.091	0.773 ±0.086	0.548 ±0.131

Table 19: Graph Theory Benchmark results on Tree graphs. All results are obtained using our codebase.

Category	Model	SP_{sssd}	MCC	Diameter	SP_{ss}	ECC
Baselines (h=1)	n=1	5.21 ±0.209	1.28 ±0.050	3.77 ±1.22	17.16 ±0.168	24.63 ±0.427
	n=5	3.34 ±0.375	0.405 ±0.089	0.504 ±0.109	7.66 ±0.325	18.11 ±1.85
	n=10	3.16 ±0.252	0.338 ±0.046	0.100 ±0.059	2.28 ±0.209	14.93 ±0.800
MeGraph(h=5) EdgePool(τ_c=2)	n=1	1.62 ±0.314	0.846 ±0.071	0.725 ±0.249	6.99 ±0.610	12.27 ±0.843
	n=5	0.83 ±0.667	0.490 ±0.118	0.084 ±0.030	1.27 ±0.442	2.87 ±0.420
MeGraph S-EdgePool Variants (h=5, n=5)	τ_c=3	0.599 ±0.200	0.483 ±0.081	0.075 ±0.012	0.497 ±0.121	0.429 ±0.105
	η_v=0.3	0.868 ±0.230	0.413 ±0.054	0.142 ±0.047	0.789 ±0.092	0.534 ±0.074
	η_v=0.3, τ_c=4	0.615 ±0.209	0.418 ±0.024	0.081 ±0.017	0.440 ±0.106	0.436 ±0.097
	η_v=0.5, τ_c=4	1.06 ±0.327	0.424 ±0.042	0.214 ±0.018	1.20 ±0.128	2.03 ±0.507
	η_v=0.3, τ_c=4 (X-Pool)	0.666 ±0.118	0.596 ±0.067	0.182 ±0.057	1.22 ±0.281	1.11 ±0.122
	η_v=0.3, τ_c=4 (w/o pw)	0.771 ±0.141	0.455 ±0.056	0.124 ±0.032	0.700 ±0.190	1.07 ±0.260
Graph-UNets	h=5,n=9,η_v=0.3,τ_c=4	0.873 ±0.247	0.667 ±0.043	0.804 ±0.284	0.606 ±0.123	1.00 ±0.221

Table 20: Graph Theory Benchmark results on Ladder graphs, all results are obtained using our codebase.

Category	Model	SP_{sssd}	MCC	Diameter	SP_{ss}	ECC
Baselines (h=1)	n=1	5.06 ±0.330	1.73 ±0.249	1.17 ±0.149	13.20 ±0.126	20.10 ±0.583
	n=5	0.692 ±0.204	0.734 ±0.106	1.39 ±0.078	5.02 ±0.876	19.81 ±0.669
	n=10	0.257 ±0.078	0.691 ±0.119	1.55 ±0.069	1.60 ±0.194	20.40 ±0.995
MeGraph(h=5) EdgePool(τ_c=2)	n=1	0.662 ±0.165	0.866 ±0.071	1.57 ±0.992	2.18 ±0.181	6.61 ±1.32
	n=5	0.251 ±0.108	0.753 ±0.091	0.175 ±0.169	0.321 ±0.058	1.18 ±0.746
MeGraph S-EdgePool Variants (h=5, n=5)	τ_c=3	0.296 ±0.070	0.754 ±0.086	0.226 ±0.069	0.228 ±0.021	0.285 ±0.069
	η_v=0.3	0.507 ±0.204	0.768 ±0.050	0.156 ±0.053	0.969 ±0.148	0.787 ±0.059
	η_v=0.3, τ_c=4	0.297 ±0.113	0.712 ±0.059	0.095 ±0.046	0.180 ±0.026	0.225 ±0.043
	η_v=0.5, τ_c=4	0.375 ±0.196	0.656 ±0.064	0.058 ±0.019	0.612 ±0.191	0.464 ±0.121
	η_v=0.3, τ_c=4 (X-Pool)	0.442 ±0.108	0.742 ±0.047	0.158 ±0.074	0.710 ±0.076	0.765 ±0.089
	η_v=0.3, τ_c=4 (w/o pw)	0.245 ±0.021	0.682 ±0.105	0.106 ±0.052	0.271 ±0.039	0.618 ±0.188
Graph-UNets	h=5,n=9,η_v=0.3,τ_c=4	0.339 ±0.11	0.797 ±0.138	0.013 ±0.005	0.287 ±0.023	0.230 ±0.054

Table 21: Graph Theory Benchmark results on Line graphs, all results are obtained using our codebase.

Category	Model	SP_{sssd}	MCC	Diameter	SP_{ss}	ECC
Baselines (h=1)	n=1	30.37 ±1.41	0.458 ±0.035	21.49 ±8.84	68.99 ±0.247	75.46 ±1.86
	n=5	10.55 ±2.40	0.019 ±0.004	9.97 ±10.85	46.39 ±3.09	78.49 ±4.38
	n=10	3.29 ±0.813	0.012 ±0.003	10.18 ±10.59	35.07 ±2.71	77.23 ±3.42
MeGraph(h=5) EdgePool(τ_c=2)	n=1	1.45 ±0.598	0.056 ±0.014	7.62 ±4.43	10.13 ±2.33	45.19 ±8.64
	n=5	0.536 ±0.149	0.016 ±0.007	0.611 ±0.238	1.06 ±0.341	14.12 ±13.82
MeGraph S-EdgePool Variants (h=5, n=5)	τ_c=3	0.349 ±0.206	0.013 ±0.003	0.724 ±0.479	0.339 ±0.102	1.15 ±0.267
	η_v=0.3	3.65 ±2.13	0.017 ±0.005	1.75 ±1.63	13.99 ±2.09	7.45 ±0.989
	η_v=0.3, τ_c=4	0.283 ±0.072	0.019 ±0.006	0.584 ±0.337	0.515 ±0.044	1.27 ±1.08
	η_v=0.5, τ_c=4	1.81 ±0.121	0.022 ±0.006	0.711 ±0.213	2.64 ±0.047	3.77 ±0.763
	η_v=0.3, τ_c=4 (X-Pool)	1.06 ±0.510	0.101 ±0.016	0.767 ±0.522	2.29 ±0.472	3.89 ±1.02
	η_v=0.3, τ_c=4 (w/o pw)	0.377 ±0.106	0.022 ±0.007	1.19 ±1.17	1.12 ±0.115	3.34 ±0.904
Graph-UNets	h=5,n=9,η_v=0.3,τ_c=4	0.426 ±0.223	0.062 ±0.008	2.89 ±1.89	0.767 ±0.129	4.78 ±1.94

Table 22: Graph Theory Benchmark results on Caterpillar graphs, all results are obtained using our codebase.

Category	Model	SP_{sssd}	MCC	Diameter	SP_{ss}	ECC
Baselines (h=1)	n=1	24.24 ±1.57	1.25 ±0.082	28.62 ±2.55	19.08 ±0.208	35.32 ±0.462
	n=5	8.32 ±2.10	0.561 ±0.070	4.59 ±0.346	9.62 ±0.357	37.01 ±1.48
	n=10	6.40 ±0.652	0.630 ±0.127	5.06 ±0.499	4.06 ±0.297	37.87 ±3.22
MeGraph(h=5) EdgePool(τ_c=2)	n=1	5.04 ±1.03	0.685 ±0.077	6.08 ±1.40	5.40 ±0.843	28.52 ±2.16
	n=5	3.44 ±1.13	0.533 ±0.064	2.00 ±1.28	0.921 ±0.149	5.20 ±1.57
MeGraph S-EdgePool Variants (h=5, n=5)	τ_c=3	2.47 ±0.529	0.607 ±0.081	0.591 ±0.172	0.574 ±0.073	1.21 ±0.148
	η_v=0.3	3.61 ±1.36	0.582 ±0.052	0.578 ±0.231	1.69 ±0.572	1.95 ±0.322
	η_v=0.3, τ_c=4	1.59 ±0.444	0.535 ±0.091	0.317 ±0.104	0.474 ±0.170	1.32 ±0.272
	η_v=0.5, τ_c=4	2.00 ±0.648	0.514 ±0.040	1.10 ±0.288	0.986 ±0.130	2.11 ±0.766
	η_v=0.3, τ_c=4 (X-Pool)	1.39 ±0.478	0.602 ±0.110	0.736 ±0.230	1.78 ±0.254	3.36 ±0.873
	η_v=0.3, τ_c=4 (w/o pw)	1.82 ±0.627	0.628 ±0.093	0.604 ±0.067	0.797 ±0.299	2.25 ±0.230
Graph-UNets	h=5,n=9,η_v=0.3,τ_c=4	1.57 ±0.670	0.679 ±0.098	3.18 ±0.583	0.976 ±0.270	3.83 ±1.06

Table 23: Graph Theory Benchmark results on Lobster graphs, all results are obtained using our codebase.

Category	Model	SP_{sssd}	MCC	Diameter	SP_{ss}	ECC
Baselines (h=1)	n=1	23.92 ±0.319	1.06 ±0.166	11.93 ±1.32	38.44 ±0.065	40.46 ±0.350
	n=5	10.89 ±1.47	0.544 ±0.067	3.66 ±0.424	20.12 ±0.105	28.81 ±1.14
	n=10	7.35 ±2.50	0.631 ±0.067	2.59 ±0.517	10.52 ±0.619	28.47 ±1.65
MeGraph(h=5) EdgePool(τ_c=2)	n=1	6.00 ±1.82	0.785 ±0.062	4.35 ±1.51	13.75 ±0.675	30.49 ±2.18
	n=5	1.93 ±0.861	0.543 ±0.073	1.07 ±0.114	2.05 ±0.393	11.39 ±5.43
MeGraph S-EdgePool Variants (h=5, n=5)	τ_c=3	2.02 ±0.791	0.447 ±0.123	0.705 ±0.133	1.66 ±0.270	2.23 ±0.378
	η_v=0.3	6.01 ±1.52	0.521 ±0.028	0.707 ±0.202	3.04 ±0.250	2.70 ±0.212
	η_v=0.3, τ_c=4	1.90 ±0.449	0.489 ±0.069	0.671 ±0.165	1.30 ±0.106	2.62 ±0.849
	η_v=0.5, τ_c=4	3.27 ±0.716	0.451 ±0.090	0.941 ±0.324	2.82 ±0.803	4.04 ±0.527
	η_v=0.3, τ_c=4 (X-Pool)	2.67 ±0.486	0.494 ±0.109	1.01 ±0.194	2.79 ±0.343	4.16 ±0.886
	η_v=0.3, τ_c=4 (w/o pw)	1.85 ±0.432	0.473 ±0.069	0.892 ±0.277	1.77 ±0.329	4.33 ±1.71
Graph-UNets	h=5,n=9,η_v=0.3,τ_c=4	4.85 ±1.48	0.782 ±0.026	3.74 ±0.361	2.96 ±0.443	4.25 ±0.544

Table 24: Graph Theory Benchmark results on Cycle graphs, all results are obtained using our codebase.

Category	Model	SP_{sssd}	MCC	Diameter	SP_{ss}	ECC
Baselines (h=1)	n=1	18.75 ±0.066	0.534 ±0.022	22.35 ±0.149	24.07 ±0.009	21.47 ±0.060
	n=5	3.39 ±0.304	0.027 ±0.001	25.11 ±0.325	12.44 ±1.05	21.81 ±0.102
	n=10	0.352 ±0.060	0.011 ±0.003	26.54 ±1.16	8.65 ±1.02	24.09 ±0.360
MeGraph(h=5) EdgePool(τ_c=2)	n=1	0.594 ±0.212	0.074 ±0.029	9.11 ±1.88	4.07 ±0.364	21.53 ±0.070
	n=5	0.060 ±0.032	0.014 ±0.003	13.44 ±6.40	0.103 ±0.016	24.05 ±0.204
MeGraph S-EdgePool Variants (h=5, n=5)	τ_c=3	0.066 ±0.036	0.015 ±0.006	0.241 ±0.049	0.090 ±0.037	0.342 ±0.186
	η_v=0.3	2.45 ±0.873	0.015 ±0.001	0.709 ±0.226	8.36 ±0.261	0.488 ±0.267
	η_v=0.3, τ_c=4	0.060 ±0.030	0.019 ±0.003	0.312 ±0.236	0.226 ±0.050	0.562 ±0.209
	η_v=0.5, τ_c=4	0.451 ±0.203	0.014 ±0.004	0.252 ±0.124	1.05 ±0.524	4.30 ±1.90
	η_v=0.3, τ_c=4 (X-Pool)	0.494 ±0.292	0.096 ±0.028	0.468 ±0.220	1.08 ±0.130	0.860 ±0.292
	η_v=0.3, τ_c=4 (w/o pw)	0.159 ±0.209	0.017 ±0.008	1.23 ±0.928	0.461 ±0.118	8.26 ±3.70
Graph-UNets	h=5,n=9,η_v=0.3,τ_c=4	0.144 ±0.073	0.035 ±0.010	3.21 ±0.893	0.439 ±0.089	5.91 ±1.31

Table 25: Graph Theory Benchmark results on Pseudotree graphs, all results are obtained using our codebase.

Category	Model	SP_{sssd}	MCC	Diameter	SP_{ss}	ECC
Baselines (h=1)	n=1	1.93 ±0.239	1.71 ±0.281	2.78 ±0.098	6.27 ±0.004	4.23 ±0.034
	n=5	0.061 ±0.024	0.942 ±0.094	1.74 ±0.299	1.54 ±0.006	4.15 ±0.086
	n=10	0.037 ±0.022	0.775 ±0.094	1.84 ±0.260	0.126 ±0.038	4.06 ±0.037
MeGraph(h=5) EdgePool(τ_c=2)	n=1	0.404 ±0.096	1.75 ±0.133	1.50 ±0.494	2.25 ±0.280	3.97 ±0.270
	n=5	0.141 ±0.022	0.999 ±0.054	1.16 ±0.069	0.148 ±0.034	3.12 ±0.202
MeGraph S-EdgePool Variants (h=5, n=5)	τ_c=3	0.130 ±0.069	0.912 ±0.073	0.669 ±0.080	0.115 ±0.015	0.797 ±0.079
	η_v=0.3	0.048 ±0.030	0.839 ±0.077	0.758 ±0.134	0.246 ±0.021	0.838 ±0.023
	η_v=0.3, τ_c=4	0.106 ±0.054	0.814 ±0.092	0.663 ±0.076	0.133 ±0.028	0.845 ±0.101
	η_v=0.5, τ_c=4	0.071 ±0.048	1.03 ±0.186	0.583 ±0.065	0.171 ±0.038	0.868 ±0.034
	η_v=0.3, τ_c=4 (X-Pool)	0.564 ±0.155	0.966 ±0.172	0.977 ±0.054	0.611 ±0.065	1.10 ±0.036
	η_v=0.3, τ_c=4 (w/o pw)	0.080 ±0.033	0.971 ±0.072	0.956 ±0.230	0.276 ±0.017	1.13 ±0.321
Graph-UNets	h=5,n=9,η_v=0.3,τ_c=4	0.467 ±0.065	1.09 ±0.072	1.71 ±0.295	0.721 ±0.092	2.25 ±0.327

Table 26: Graph Theory Benchmark results on Geo graphs, all results are obtained using our codebase.

Category	Model	SP_{sssd}	MCC	Diameter	SP_{ss}	ECC
Baselines (h=1)	n=1	5.79 ±0.630	0.424 ±0.023	11.85 ±0.391	12.49 ±0.035	14.82 ±0.056
	n=5	1.02 ±0.772	0.407 ±0.040	8.37 ±0.468	5.10 ±0.435	14.33 ±0.079
	n=10	0.304 ±0.125	0.404 ±0.061	9.41 ±0.759	0.803 ±0.162	14.33 ±0.136
MeGraph(h=5) EdgePool(τ_c=2)	n=1	1.60 ±0.880	0.347 ±0.033	10.17 ±2.04	4.87 ±0.777	11.91 ±0.451
	n=5	0.232 ±0.061	0.273 ±0.018	2.70 ±0.288	0.575 ±0.127	6.92 ±2.36
MeGraph S-EdgePool Variants (h=5, n=5)	τ_c=3	0.188 ±0.100	0.288 ±0.020	2.04 ±0.225	0.562 ±0.186	2.42 ±0.333
	η_v=0.3	1.38 ±0.617	0.330 ±0.025	4.40 ±1.15	1.37 ±0.083	5.45 ±0.465
	η_v=0.3, τ_c=4	0.230 ±0.070	0.231 ±0.034	1.99 ±0.549	0.454 ±0.057	2.69 ±0.369
	η_v=0.5, τ_c=4	0.374 ±0.148	0.368 ±0.043	3.95 ±0.319	0.777 ±0.122	4.61 ±0.717
	η_v=0.3, τ_c=4 (X-Pool)	1.04 ±0.502	0.362 ±0.031	2.32 ±0.440	2.37 ±0.260	5.08 ±0.737
	η_v=0.3, τ_c=4 (w/o pw)	0.233 ±0.046	0.261 ±0.035	2.58 ±0.617	1.09 ±0.226	4.85 ±0.805
Graph-UNets	h=5,n=9,η_v=0.3,τ_c=4	1.49 ±0.451	0.400 ±0.020	4.63 ±0.647	2.42 ±0.458	7.36 ±1.62

Table 27: Graph Theory Benchmark results on BA graphs, all results are obtained using our codebase.

Category	Model	SP_{sssd}	MCC	Diameter	SP_{ss}	ECC
Baselines (h=1)	n=1	0.004 ±0.001	2.81 ±0.142	0.092 ±0.021	–	0.128 ±0.006
	n=5	0.007 ±0.002	3.65 ±0.660	0.098 ±0.014	–	0.091 ±0.011
	n=10	0.011 ±0.006	3.72 ±0.376	0.122 ±0.038	–	0.080 ±0.004
MeGraph(h=5) EdgePool(τ_c=2)	n=1	0.006 ±0.004	2.00 ±0.380	0.101 ±0.020	–	0.084 ±0.017
	n=5	0.003 ±0.001	2.00 ±0.240	0.104 ±0.011	–	0.052 ±0.010
MeGraph S-EdgePool Variants (h=5, n=5)	τ_c=3	0.007 ±0.003	1.77 ±0.403	0.089 ±0.008	–	0.126 ±0.027
	η_v=0.3	0.013 ±0.004	1.67 ±0.333	0.084 ±0.008	–	0.086 ±0.005
	η_v=0.3, τ_c=4	0.011 ±0.005	1.42 ±0.252	0.073 ±0.015	–	0.163 ±0.007
	η_v=0.5, τ_c=4	0.008 ±0.004	1.71 ±0.403	0.074 ±0.009	–	0.156 ±0.021
	η_v=0.3, τ_c=4 (X-Pool)	0.009 ±0.003	1.22 ±0.242	0.088 ±0.021	–	0.076 ±0.006
	η_v=0.3, τ_c=4 (w/o pw)	0.009 ±0.003	1.42 ±0.209	0.068 ±0.017	–	0.068 ±0.017
Graph-UNets	h=5,n=9,η_v=0.3,τ_c=4	0.024 ±0.009	2.84 ±0.777	0.091 ±0.01	–	0.179 ±0.0227

Table 28: Graph Theory Benchmark results on mixed, ER, Caveman and Star graphs, all results are obtained using our codebase.

Category	Model	MCC				ECC	
		mix	ER	Caveman	Star	mix	ER
Baselines (h=1)	n=1	3.46 ±0.211	2.91 ±0.206	0.015 ±0.004	0.144 ±0.031	0.316 ±0.003	0.346 ±0.006
	n=5	3.29 ±0.261	3.35 ±0.205	0.014 ±0.003	0.078 ±0.021	0.228 ±0.008	0.289 ±0.008
	n=10	3.51 ±0.323	3.53 ±0.375	0.018 ±0.006	0.065 ±0.005	0.212 ±0.008	0.414 ±0.102
MeGraph(h=5) EdgePool(τ_c=2)	n=1	1.25 ±0.167	0.749 ±0.058	0.018 ±0.005	0.135 ±0.055	0.150 ±0.011	0.320 ±0.071
	n=5	1.11 ±0.143	0.723 ±0.073	0.017 ±0.005	0.052 ±0.017	0.125 ±0.010	0.345 ±0.064
MeGraph S-EdgePool Variants (h=5, n=5)	τ_c=3	1.07 ±0.034	0.714 ±0.039	0.017 ±0.002	0.072 ±0.016	0.137 ±0.013	0.232 ±0.035
	η_v=0.3	0.908 ±0.153	0.627 ±0.090	0.026 ±0.007	0.125 ±0.026	0.128 ±0.014	0.248 ±0.012
	η_v=0.3, τ_c=4	1.10 ±0.085	0.709 ±0.092	0.019 ±0.004	0.073 ±0.012	0.129 ±0.009	0.224 ±0.053
	η_v=0.5, τ_c=4	1.12 ±0.219	0.722 ±0.128	0.026 ±0.008	0.058 ±0.010	0.147 ±0.017	0.219 ±0.042
	η_v=0.3, τ_c=4 (X-Pool)	1.01 ±0.166	0.838 ±0.078	0.029 ±0.007	0.107 ±0.021	0.119 ±0.008	0.213 ±0.027
	η_v=0.3, τ_c=4 (w/o pw)	1.13 ±0.059	0.622 ±0.073	0.019 ±0.003	0.075 ±0.015	0.126 ±0.016	0.307 ±0.062
Graph-UNets	h=5,n=9,η_v=0.3,τ_c=4	1.06 ±0.171	0.859 ±0.092	0.041 ±0.007	0.057 ±0.010	0.153 ±0.012	0.34 5±0.133