Hierarchical Graph Generation with K^2–trees

Yunhui Jang [1] **Dongwoo Kim** [1] **Sungsoo Ahn** [1]

Abstract

Generating graphs from a target distribution is a significant challenge across many domains. In this work, we introduce a novel graph generation method leveraging K^2–tree representation which was originally designed for lossless graph compression. Our motivation stems from the ability of the K^2–tree to enable compact generation while concurrently capturing the inherent hierarchical structure of a graph. In addition, we make further contributions by (1) presenting a sequential K^2–tree representation that incorporates pruning, flattening, and tokenization processes and (2) introducing a Transformer-based architecture designed to generate the sequence by incorporating a specialized tree positional encoding scheme. Finally, we extensively evaluate our algorithm on four general and two molecular graph datasets to confirm its superiority for graph generation.

1. Introduction

Generating graph-structured data is a challenging problem in numerous fields. Recently, deep generative models have demonstrated significant potential in addressing this challenge (Maziarka et al., 2020; Simonovsky & Komodakis, 2018; Madhawa et al., 2019; Liu et al., 2021; Jo et al., 2022; Vignac et al., 2022). The deep graph generative models can be categorized into three types by their graph representation: adjacency matrix (Simonovsky & Komodakis, 2018; Madhawa et al., 2019; Liu et al., 2021; Maziarka et al., 2020), string representation (Ahn et al., 2022; Goyal et al., 2020; Krenn et al., 2019; Segler et al., 2018), and motif-based representation (Jin et al., 2018; 2020).

Despite the absence of a consensus on the optimal graph representation, the evolution of these models is driven by two primary factors: the need for compactness and the presence of hierarchy within graphs. Compactness reduces graph generation complexity and simplifies the search space over graphs. For example, string and motif representations provide compact representations compared to adjacency matrices. Otherwise, hierarchy is an inherent characteristic in many types of graphs. While motif representations address hierarchy in molecular graphs, their application is restricted due to the fixed vocabulary of motifs in a specific domain and their inability to generalize to the hierarchical structure of general graphs.

Contribution. In this paper, we propose a novel graph generation framework, coined **H**ierarchical **G**raph **G**eneration with K^2–**T**ree (HGGT), which is compact, hierarchical, able to represent attributed graphs, and does not require domain-specific rules.

To this end, We design a sequential representation that is more compact than the K^2–tree, minimizing the number of decisions the autoregressive model requires. Our method involves a two-stage procedure: (1) pruning the K^2-Tree to eliminate redundancy from the symmetric adjacency matrix in undirected graphs, (2) flattening and tokenizing the K^2-Tree into a sequence to minimize the decision-making requirements of graph generation further.

The Transformer architecture (Vaswani et al., 2017) is employed to generate a K^2–tree representation of a graph. To effectively incorporate positional information of each tree-node, we devise a new positional encoding scheme tailored to the K^2–tree structure. The positional information of a tree-node is represented by its pathway from the root node, enabling full reconstruction of the K^2-Tree from just the positional information. The validity of our algorithm is tested on popular graph generation benchmarks across six graph datasets, and HGGT outperformed the baselines.

2. K^2–tree Representation of a Graph

We consider the K^2–tree representation (Brisaboa et al., 2009) $(\mathcal{T}, \mathcal{X})$ of an adjacency matrix A as a K^2-ary tree $\mathcal{T} = (\mathcal{V}, \mathcal{E})$ associated with binary node attributes $\mathcal{X} = \{x_u : u \in \mathcal{V}\}$. Every non-root node is uniquely indexed as (i, j)-th child of its parent node for some $i, j \in \{1, \ldots, K\}$. The tree $\mathcal{T}$ is ordered so that every (i, j)-th child node is ranked $K(i-1)+j$ among its siblings. Then the K^2–tree

[1]Graduate school of Artificial Intelligence, POSTECH, Pohang, South Korea. Correspondence to: Sungsoo Ahn <sungsoo.ahn@postech.ac.kr>.

Accepted to ICML workshop on Structured Probabilistic Inference & Generative Modeling, Honolulu, Hawaii, USA. PMLR 202, 2023.

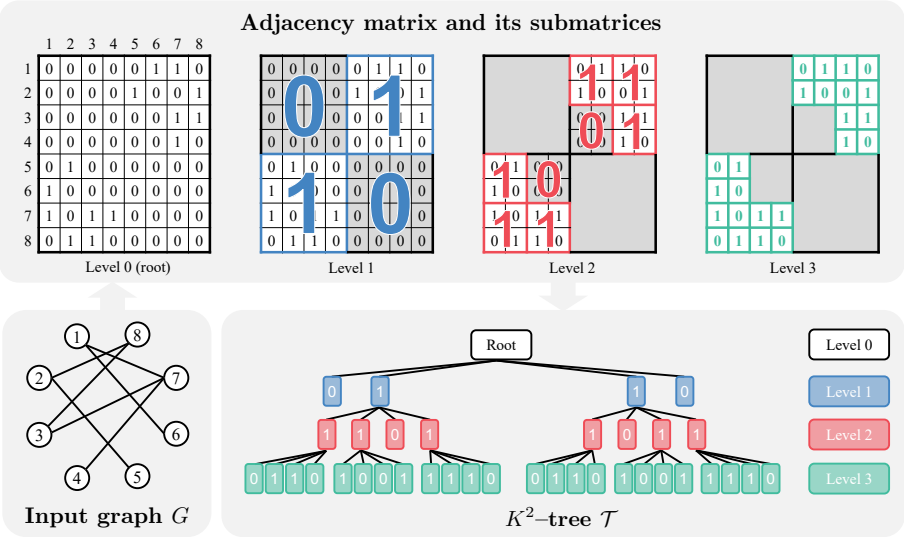


Figure 1. K^2**–tree with** $K = 2$**.** The K^2–tree describes the hierarchy of the adjacency matrix iteratively being partitioned to $K \times K$ submatrices. It is compact due to summarizing any zero-filled submatrix with size larger than 1×1 (shaded in grey) by a non-leaf node u with label $x_u = 0$.

satisfies the following conditions:

- Each tree-node u is associated with a submatrix $A^{(u)}$ of the adjacency matrix A.
- If the submatrix $A^{(u)}$ for a tree-node u is filled only with zeros, $x_u = 0$. Otherwise, $x_u = 1$.
- A tree-node u is a leaf node if and only if $x_u = 0$ or the matrix $A^{(u)}$ is a 1×1 matrix.
- Let $B_{1,1}, \ldots, B_{K,K}$ denote the $K \times K$ partitioning of the matrix $A^{(u)}$ with i, j corresponding to row- and column-wise order, respectively. The child nodes $v_{1,1}, \ldots, v_{K,K}$ of the tree-node u are associated with the submatrices $B_{1,1}, \ldots, B_{K,K}$, respectively.

The illustration of the K^2–tree can be found in Figure 1. The generated K^2–tree is a compact description of graph G as any non-leaf node u with $x_u = 0$ summarizes a large submatrix filled only with zeros. In the worst-case scenario, the size of the K^2–tree is $MK^2(\log_{K^2}(N^2/M) + O(1))$ (Brisaboa et al., 2009), where N and M denote the number of nodes and edges in the original graph, respectively. This constitutes a significant improvement over the N^2 size of the full adjacency matrix.

Additionally, the K^2–tree is a hierarchical representation of the original graph, ensuring that (1) each node within the tree represents the connectivity between a specific set of nodes, and (2) nodes closer to the root correspond to a larger set of nodes. We emphasize that the tree-nodes are associated with submatrices overlapping with the diagonal of the original adjacency matrix when they indicate intra-connectivity within a group of nodes. In contrast, the remaining tree-nodes describe the inter-connectivity between two distinct sets of nodes. We also describe the detailed algorithms for constructing a K^2–tree from a given graph G and recovering a graph from the K^2–tree in Appendices A and B, respectively.

3. Hierarchical Graph Generation with K^2–trees

3.1. Sequential K^2–tree representation

Here, we propose an algorithm to flatten the K^2–tree into a sequence as illustrated in Figure 2.

Pruning the K^2**–tree.** To obtain the pruned K^2–tree, we identify and eliminate redundant tree-nodes due to the symmetry of the adjacency matrix for undirected graphs. In particular, such tree-nodes are associated with submatrices positioned above the diagonal since they mirror the counterparts below it.

We now describe a formula to identify redundant tree-nodes using the position of a submatrix $A^{(u)}$, tied to a specific tree-node u, within the adjacency matrix A. To this end, we consider a downward path $v_0, v_1, \ldots, v_L$ from the root node $r = v_0$ to the tree-node $u = v_L$ as description of the tree-node position. We also let (i_{v_ℓ}, j_{v_ℓ}) denote the order of v_ℓ among its siblings. Hence, the tree-node position can be represented as $\mathrm{pos}(u) = ((i_{v_1}, j_{v_1}), \ldots, (i_{v_L}, j_{v_L}))$. Also, the location of the submatrix $A^{(u)}$ is derived as the $(p_u, q_u) = (\sum_{\ell=1}^{L} K^{L-\ell}(i_{v_\ell} - 1) + 1, \sum_{\ell=1}^{L} K^{L-\ell}(j_{v_\ell} - 1) + 1)$-th element with respect to the $K^L \times K^L$ partition

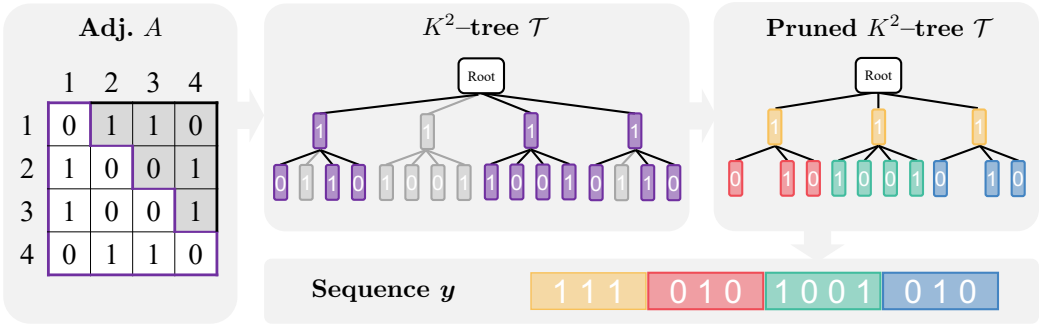


Figure 2. **Illustration of the sequential representation for K^2–tree.** The shaded parts of the adjacency matrix A and the K^2–tree $\mathcal{T}$ denote redundant parts, which are further pruned, while the purple-colored parts of A and $\mathcal{T}$ denote non-redundant parts. Also, same-colored tree-nodes of pruned K^2–tree are grouped and tokenized into the same colored parts of the sequence $\boldsymbol{y}$.

of the adjacency matrix A, as described in Appendix B. As a result, we eliminate any tree-node associated with a submatrix above the diagonal, i.e., we remove tree-node u when $p_u < q_u$. Consequently, the pruned K^2–tree maintains only tree-nodes associated with submatrices devoid of redundant nodes.

Flattening and tokenization of the pruned K^2–tree. Next, we explain how to obtain a sequential representation of the pruned K^2–tree based on flattening and tokenization. Our idea is to flatten a K^2–tree as a sequence of tree-node attributes $\{x_u : u \in \mathcal{V}\}$ using breadth-first traversal and then to tokenize the sequence by grouping the tree-nodes that share the same parent node, i.e., sibling nodes. The detailed description is described in Appendix C. We also extend our HGGT to graphs with node and edge-wise features, e.g., molecular graphs, and the detailed description is in Appendix D.

3.2. Generating K^2–tree with Transformer and K^2–tree positional encoding

Transformer with K^2–tree positional encoding. We first propose the Transformer (Vaswani et al., 2017) architecture to parameterize the distribution $p_\theta(y_t|y_{t-1}, \dots, y_1)$. To this end, we base our architecture on the Transformer architecture used for the autoregressive generation of sequential data. We use the tree-positional encodings for each time-step t to incorporate the structure of K^2–tree during generation.

As outlined in Section 3.1, the node attributes in y_t are associated with child nodes of a particular tree-node u. To enhance the input y_t, we incorporate the positional encoding for u. We update the input feature y_t through the addition of positional encoding, which is represented as $\text{PE}(u) = \sum_{\ell=1}^{L} \text{Embedding}_\ell(i_{v_\ell}, j_{v_\ell})$, where $((i_{v_1}, j_{v_1}), \dots, (i_{v_L}, j_{v_L}))$ are the sequence of orders of a downward path from the root node r to the tree-node u.

Constructing K^2–tree from the sequential representation. We next explain the algorithm to recover a K^2–tree from its sequential representation $\boldsymbol{y}$. In particular, we generate the K^2–tree simultaneously with the sequence to incorporate the tree information for each step of the autoregressive generation. The algorithm begins with an empty tree containing only a root node and iteratively expands each "frontier" node based on the sequence of the decisions made by the generative model. To facilitate a breadth-first expansion approach, the algorithm utilizes a first-in-first-out (FIFO) queue, which contains tree-node candidates to be expanded.

To be specific, our algorithm initializes a K^2–tree $\mathcal{T} = (\{r\}, \emptyset)$ with the root node r associated with the node attribute $x_r = 1$. It also initializes the FIFO queue $\mathcal{Q}$ with the root node r. Then at each t-th iteration, our algorithm expands the tree-node u popped from the queue $\mathcal{Q}$ using the token y_t. To be specific, for each tree-node attribute x in y_t, our algorithm adds a child node v with $x_v = x$. If $x = 1$ and the size of $A^{(v)}$ is larger than 1×1, the child node v is inserted into the queue Q.

4. Experiment

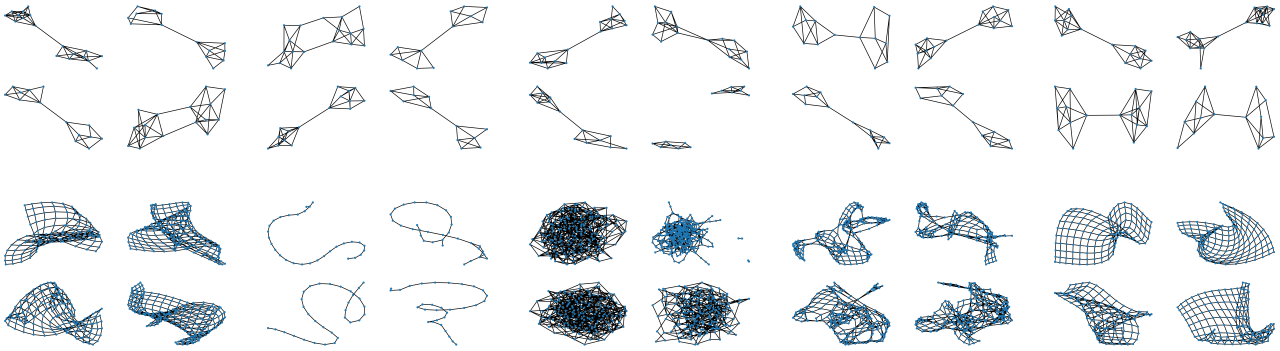
4.1. Generic graph generation

Experimental setup. We first validate the general graph generation performance of our HGGT on four datasets: (1) **Community-small**, (2) **Planar**, (3) **Enzymes** (Schomburg et al., 2004), (4) **Grid**. We adopt maximum mean discrepancy (MMD) as our evaluation metric to compare three graph property distributions between generated graphs and test graphs: degree, clustering coefficient, and 4-node-orbit counts. The baselines are described in Table 1 and detailed descriptions of our experimental setup are in Appendix E.

Results. We provide experimental results are in Table 1. We observe that our HGGT clearly outperforms all the baselines on all the datasets. This verifies our model's ability to effectively capture the structural information of graphs. In particular, we observe how the performance of HGGT is

Table 1. **Generic graph generation performance.** For each metric, the best is highlighted in **bold** and the second-best is underlined.

	Community-small			Planar			Enzymes			Grid		
	$12 \leq \|V\| \leq 20$			$\|V\| = 64$			$10 \leq \|V\| \leq 125$			$100 \leq \|V\| \leq 400$		
Method	Deg.	Clus.	Orb.	Deg.	Clus.	Orb.	Deg.	Clus.	Orb.	Deg.	Clus.	Orb.
GraphVAE (Simonovsky & Komodakis, 2018)	0.350	0.980	0.540	-	-	-	1.369	0.629	0.191	1.619	**0.000**	0.919
GraphRNN (You et al., 2018)	0.080	0.120	0.040	0.005	0.278	1.254	0.017	0.062	0.046	0.064	0.043	0.021
GNF (Liu et al., 2019)	0.200	0.200	0.110	-	-	-	-	-	-	-	-	-
GRAN (Liao et al., 2019)	-	-	-	0.001	0.043	0.001	-	-	-	0.001	0.004	0.002
EDP-GNN (Niu et al., 2020)	0.053	0.144	0.026	-	-	-	0.023	0.268	0.082	0.455	0.238	0.328
GraphGen (Goyal et al., 2020)	0.075	0.065	0.014	1.762	1.423	1.640	0.146	0.079	0.054	1.550	0.017	0.860
GraphAF (Shi et al., 2020)	0.180	0.200	0.020	-	-	-	1.669	1.283	0.266	-	-	-
GraphDF (Luo et al., 2021)	0.060	0.120	0.030	-	-	-	1.503	1.283	0.266	-	-	-
SPECTRE (Martinkus et al., 2022)	-	-	-	0.010	0.067	0.010	-	-	-	-	-	-
GDSS (Jo et al., 2022)	0.045	0.086	0.007	-	-	-	0.026	0.061	0.009	0.111	0.005	0.070
DiGress (Vignac et al., 2022)	0.012	0.025	0.002	**0.000**	0.002	0.008	0.011	0.039	0.010	0.016	**0.000**	0.004
GDSM (Luo et al., 2022)	0.011	0.015	**0.001**	-	-	-	0.013	0.088	0.010	0.002	**0.000**	**0.000**
HGGT (ours)	**0.001**	**0.006**	0.003	**0.000**	**0.001**	**0.000**	**0.005**	**0.017**	**0.000**	**0.000**	**0.000**	**0.000**



(a) Train (b) GraphGen (Goyal et al., 2020) (c) GDSS (Jo et al., 2022) (d) DiGress (Vignac et al., 2022) (e) HGGT (ours)

Figure 3. Generated samples for Community-small (top), and Grid (bottom) datasets.

extraordinary for Grid. We hypothesize that HGGT achieves high performance due to capturing the coarse-grained structure of grid graphs. In Figure 3, we also present examples of generated graphs to demonstrate the efficacy of our proposed algorithm, HGGT, in capturing the structural information of training graphs more accurately compared to existing graph generative models. We additionally provide visualizations of generated samples in Appendix G.

4.2. Molecular graph generation

Experimental setup. We use two molecular datasets: **QM9** (Ramakrishnan et al., 2014) and **ZINC250k** (Irwin et al., 2012). We evaluate 10,000 generated molecules using six metrics: (a) validity (Val.), (b) neighborhood subgraph pairwise distance kernel (NSPDK), and (c) Frechet ChemNet Distance (FCD) (Preuer et al., 2018). The baselines are described in Table 2 and we provide additional details for the experimental setup in Appendix E.

Results. The experimental results are reported in Table 2. We observe that our HGGT showed competitive results on all the baselines on most of the metrics. This also confirms the effectiveness of our modification of HGGT to generate the featured graph. In particular, for the ZINC250k dataset, we observe a large gap between our method and the baselines for NSPDK and FCD scores. This verifies the ability of our model to be trained on large molecular graphs. We additionally provide visualizations of generated molecules in Appendix G and additional metrics in Appendix H.

Table 2. **Molecular graph generation performance.** The best is highlighted in **bold** and the second-best is underlined.

	QM9			ZINC250k		
Method	Val. ↑	NSPDK ↓	FCD ↓	Val. ↑	NSPDK ↓	FCD ↓
EDP-GNN (Niu et al., 2020)	47.52	0.005	2.68	82.97	0.049	16.74
MoFlow (Zang & Wang, 2020)	91.36	0.017	4.47	63.11	0.046	20.93
GraphAF (Shi et al., 2020)	74.43	0.020	5.27	68.47	0.044	16.02
GraphDF (Luo et al., 2021)	93.88	0.064	10.93	90.61	0.177	33.55
GraphEBM (Liu et al., 2021)	8.22	0.030	6.14	5.29	0.212	35.47
GDSS (Jo et al., 2022)	95.72	0.003	2.90	97.01	0.019	14.66
DiGress (Vignac et al., 2022)	99.01	0.001	**0.25**	**100**	0.042	16.54
GDSM (Luo et al., 2022)	**99.90**	0.003	2.65	92.70	0.017	12.96
HGGT (ours)	99.22	**0.000**	0.40	92.87	**0.001**	**1.93**

5. Acknowledgements

This work was partly supported by Basic Science Research Program through the National Research Foundation of Korea(NRF) funded by the Ministry of Education(2022R1A6A1A03052954, NRF-2021R1C1C1011375) and Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government(MSIT) (No.2019-0-01906, Artificial Intelligence Graduate School Program(POSTECH)).

References

Ahn, S., Chen, B., Wang, T., and Song, L. Spanning tree-based graph generation for molecules. In *International Conference on Learning Representations*, 2022. URL https://openreview.net/forum?id=w60btE_8T2m.

Besta, M. and Hoefler, T. Survey and taxonomy of lossless graph compression and space-efficient graph representations. *arXiv preprint arXiv:1806.01799*, 2018.

Bouritsas, G., Loukas, A., Karalias, N., and Bronstein, M. Partition and code: learning how to compress graphs. *Advances in Neural Information Processing Systems*, 34: 18603–18619, 2021.

Brisaboa, N. R., Ladra, S., and Navarro, G. k2-trees for compact web graph representation. In *SPIRE*, volume 9, pp. 18–30. Springer, 2009.

Cuthill, E. and McKee, J. Reducing the bandwidth of sparse symmetric matrices. In *Proceedings of the 1969 24th national conference*, pp. 157–172, 1969.

Goyal, N., Jain, H. V., and Ranu, S. Graphgen: a scalable approach to domain-agnostic labeled graph generation. In *Proceedings of The Web Conference 2020*, pp. 1253–1263, 2020.

Guo, M., Thost, V., Li, B., Das, P., Chen, J., and Matusik, W. Data-efficient graph grammar learning for molecular generation. *arXiv preprint arXiv:2203.08031*, 2022.

Irwin, J. J., Sterling, T., Mysinger, M. M., Bolstad, E. S., and Coleman, R. G. Zinc: a free tool to discover chemistry for biology. *Journal of chemical information and modeling*, 52(7):1757–1768, 2012.

Jin, W., Barzilay, R., and Jaakkola, T. Junction tree variational autoencoder for molecular graph generation. In *International conference on machine learning*, pp. 2323–2332. PMLR, 2018.

Jin, W., Barzilay, R., and Jaakkola, T. Hierarchical generation of molecular graphs using structural motifs. In *International conference on machine learning*, pp. 4839–4848. PMLR, 2020.

Jo, J., Lee, S., and Hwang, S. J. Score-based generative modeling of graphs via the system of stochastic differential equations. In *International Conference on Machine Learning*, pp. 10362–10383. PMLR, 2022.

Krenn, M., Häse, F., Nigam, A., Friederich, P., and Aspuru-Guzik, A. SELFIES: a robust representation of semantically constrained graphs with an example application in chemistry. *arXiv preprint arXiv:1905.13741*, 2019.

Liao, R., Li, Y., Song, Y., Wang, S., Hamilton, W., Duvenaud, D. K., Urtasun, R., and Zemel, R. Efficient graph generation with graph recurrent attention networks. *Advances in neural information processing systems*, 32, 2019.

Liu, J., Kumar, A., Ba, J., Kiros, J., and Swersky, K. Graph normalizing flows. *Advances in Neural Information Processing Systems*, 32, 2019.

Liu, M., Yan, K., Oztekin, B., and Ji, S. Graphebm: Molecular graph generation with energy-based models. *arXiv preprint arXiv:2102.00546*, 2021.

Luo, T., Mo, Z., and Pan, S. J. Fast graph generative model via spectral diffusion. *arXiv preprint arXiv:2211.08892*, 2022.

Luo, Y., Yan, K., and Ji, S. Graphdf: A discrete flow model for molecular graph generation. In *International Conference on Machine Learning*, pp. 7192–7203. PMLR, 2021.

Madhawa, K., Ishiguro, K., Nakago, K., and Abe, M. Graphnvp: An invertible flow model for generating molecular graphs. *arXiv preprint arXiv:1905.11600*, 2019.

Martinkus, K., Loukas, A., Perraudin, N., and Wattenhofer, R. Spectre: Spectral conditioning helps to overcome the expressivity limits of one-shot graph generators. In *International Conference on Machine Learning*, pp. 15159–15179. PMLR, 2022.

Maziarka, Ł., Pocha, A., Kaczmarczyk, J., Rataj, K., Danel, T., and Warchoł, M. Mol-cyclegan: a generative model for molecular optimization. *Journal of Cheminformatics*, 12(1):1–18, 2020.

Niu, C., Song, Y., Song, J., Zhao, S., Grover, A., and Ermon, S. Permutation invariant graph generation via score-based generative modeling. In *International Conference on Artificial Intelligence and Statistics*, pp. 4474–4484. PMLR, 2020.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

Preuer, K., Renz, P., Unterthiner, T., Hochreiter, S., and Klambauer, G. Fréchet chemnet distance: a metric for generative models for molecules in drug discovery. *Journal of chemical information and modeling*, 58(9):1736–1741, 2018.

Raghavan, S. and Garcia-Molina, H. Representing web graphs. In *Proceedings 19th International Conference on Data Engineering (Cat. No. 03CH37405)*, pp. 405–416. IEEE, 2003.

Ramakrishnan, R., Dral, P. O., Rupp, M., and Von Lilienfeld, O. A. Quantum chemistry structures and properties of 134 kilo molecules. *Scientific data*, 1(1):1–7, 2014.

Schomburg, I., Chang, A., Ebeling, C., Gremse, M., Heldt, C., Huhn, G., and Schomburg, D. Brenda, the enzyme database: updates and major new developments. *Nucleic acids research*, 32(suppl_1):D431–D433, 2004.

Segler, M. H., Kogej, T., Tyrchan, C., and Waller, M. P. Generating focused molecule libraries for drug discovery with recurrent neural networks. *ACS central science*, 4 (1):120–131, 2018.

Shi, C., Xu, M., Zhu, Z., Zhang, W., Zhang, M., and Tang, J. Graphaf: a flow-based autoregressive model for molecular graph generation. *arXiv preprint arXiv:2001.09382*, 2020.

Simonovsky, M. and Komodakis, N. Graphvae: Towards generation of small graphs using variational autoencoders. In *Artificial Neural Networks and Machine Learning–ICANN 2018: 27th International Conference on Artificial Neural Networks, Rhodes, Greece, October 4-7, 2018, Proceedings, Part I 27*, pp. 412–422. Springer, 2018.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

Vignac, C., Krawczuk, I., Siraudin, A., Wang, B., Cevher, V., and Frossard, P. Digress: Discrete denoising diffusion for graph generation. *arXiv preprint arXiv:2209.14734*, 2022.

Weininger, D. Smiles, a chemical language and information system. 1. introduction to methodology and encoding rules. *Journal of chemical information and computer sciences*, 28(1):31–36, 1988.

Yang, S., Hwang, D., Lee, S., Ryu, S., and Hwang, S. J. Hit and lead discovery with explorative rl and fragment-based molecule generation. *Advances in Neural Information Processing Systems*, 34:7924–7936, 2021.

You, J., Ying, R., Ren, X., Hamilton, W., and Leskovec, J. Graphrnn: Generating realistic graphs with deep auto-regressive models. In *International conference on machine learning*, pp. 5708–5717. PMLR, 2018.

Zang, C. and Wang, F. Moflow: an invertible flow model for generating molecular graphs. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pp. 617–626, 2020.

A. Construction of a K^2–tree from the graph

Algorithm 1 K^2–tree construction

Input:Adjacency matrix A and partitioning factor K.
Initialize the tree $\mathcal{T} \leftarrow (\mathcal{V}, \mathcal{E})$ with $\mathcal{V} = \emptyset, \mathcal{E} = \emptyset$. ▷ K^2–tree.
Initialize an empty queue Q. ▷ Candidates to be expanded into child nodes.
Set $\mathcal{V} \leftarrow \mathcal{V} \cup \{r\}$, $x_r \leftarrow 1$ and let $A^{(r)} \leftarrow A$. Insert r into the queue Q. ▷ Add root node r.
while $Q \neq \emptyset$ **do**
    Pop u from $\mathcal{Q}$.
    if $x_u = 0$ **then** ▷ Condition for not expanding the node u.
        Go to line 4.
    end if
    Update $s \leftarrow \dim(A^{(u)})/K$
    for $i = 1, \ldots, K$ **do** ▷ Row-wise indices.
        for $j = 1, \ldots, K$ **do** ▷ Column-wise indices.
            Set $B_{i,j} \leftarrow A^{(u)}[(i-1)s : is, (j-1)s : js]$.
            ▷ Operation to obtain $s \times s$ submatrix $B_{i,j}$ of $A^{(u)}$.
            If $B_{i,j}$ is filled with zeros, set $x_v \leftarrow 1$. Otherwise, set $x_v \leftarrow 0$.
            ▷ Update tree-node attribute.
            If $\dim(v_{i,j}) > 1$, update $\mathcal{Q} \leftarrow v_{i,j}$.
        end for
    end for
    Set $\mathcal{V} \leftarrow \mathcal{V} \cup \{v_{1,1}, \ldots, v_{K,K}\}$. ▷ Update tree nodes.
    Set $\mathcal{E} \leftarrow \mathcal{E} \cup \{(u, v_{1,1}), \ldots, (u, v_{K,K})\}$. ▷ Update tree edges.
end while
Output: K^2–tree $(\mathcal{T}, \mathcal{X})$ where $\mathcal{X} = \{x_u : u \in \mathcal{V}\}$.

In this section, we explain our algorithm to construct a K^2–tree $(\mathcal{T}, \mathcal{X})$ from a given graph $G = A$ where G is a symmetric non-featured graph and A is an adjacency matrix. Note that he K^2-ary tree $\mathcal{T} = (\mathcal{V}, \mathcal{E})$ is associated with binary node attributes $\mathcal{X} = \{x_u : u \in \mathcal{V}\}$. In addition, let $\dim(A)$ to denote the number of rows(or columns) n of the square matrix $A \in \{0, 1\}^{n \times n}$. We describe the full procedure in Algorithm 1.

B. Constructing a graph from the K^2–tree

Algorithm 2 Graph G construction

Input: K^2–tree $(\mathcal{T}, \mathcal{X})$ and partitioning factor K.
Set $m \leftarrow K^{D_\mathcal{T}}$. ▷ Full adjacency matrix size.
Initialize $A \in \{0,1\}^{m \times m}$ with zeros.
for $u \in \mathcal{L}$ **do** ▷ For each leaf node with $x_u = 1$.
  $\text{pos}(u) = ((i_{v_1}, j_{v_1}), \ldots, (i_{v_L}, j_{v_L}))$. ▷ Position of node u.
  $(p_u, q_u) = (\sum_{\ell=1}^{L} K^{L-\ell}(i_{v_\ell} - 1) + 1, \sum_{\ell=1}^{L} K^{L-\ell}(j_{v_\ell} - 1) + 1)$. ▷ Location of node u.
  Set $A_{p_u, q_u} \leftarrow 1$.
end for
Output: Adjacency matrix A.

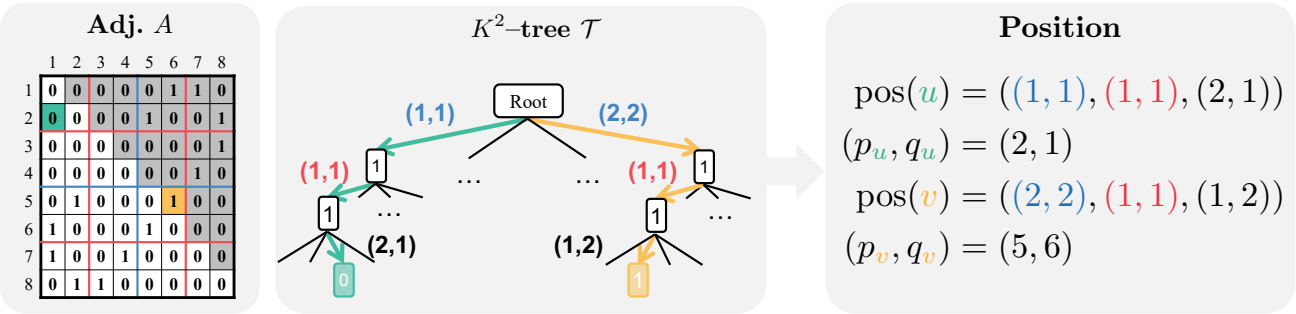


Figure 4. **Illustration of the tree-node positions of K^2–tree.** The shaded parts of the adjacency matrix denote redundant parts, e.g., $p_u < q_u$. Additionally, colored elements correspond to tree-nodes of the same color and the same-colored tree-edges signify the root-to-target downward path. Blue and red tuples denote the order in the first and second levels, respectively. The tree node u is non-redundant as $p_u > q_u$ while v is redundant as $p_v < q_v$.

We next describe the algorithm to generate a graph $G = A$ given the K^2–tree $(\mathcal{T}, \mathcal{X})$ with tree depth $D_\mathcal{T}$. Let $\mathcal{L} \subset \mathcal{V}$ be the set of leaf nodes in K^2–tree with node attributes 1. Note that we represent the tree-node position of $u \in \mathcal{V}$ as $\text{pos}(u) = ((i_{v_1}, j_{v_1}), \ldots, (i_{v_L}, j_{v_L}))$ based on a downward path $v_0, v_1, \ldots, v_L$ from the root node $r = v_0$ to the tree-node $u = v_L$. In addition, the location of corresponding submatrix $A^{(u)}$ is denoted as $(p_u, q_u) = (\sum_{\ell=1}^{L} K^{L-\ell}(i_{v_\ell} - 1) + 1, \sum_{\ell=1}^{L} K^{L-\ell}(j_{v_\ell} - 1) + 1)$ in as described in Section 3.1. The position and location of tree-nodes are illustrated in Figure 4 and we describe the full procedure as in Algorithm 2.

C. Details of flattening and tokenization

In this section, we describe details of HGGT in flattening and tokenization of the pruned K^2–tree.

To flatten a K^2–tree as a sequence, we denote the sequence of tree-nodes obtained from a breadth-first traversal of non-root tree-nodes in the K^2–tree as $u_1, \ldots, u_{|\mathcal{V}|-1}$, and the corresponding sequence of node attributes as $\boldsymbol{x} = (x_1, \ldots, x_{|\mathcal{V}|-1})$. It's important to note that sibling nodes sharing the same parent appear sequentially in the breadth-first traversal.

Next, by grouping the sibling nodes, we tokenize the sequence $\boldsymbol{x}$. As a result, we obtain a sequence $\boldsymbol{y} = (y_1, \ldots, y_T)$ where each element is a token representing a group of attributes associated with sibling nodes. For example, the t-th token corresponding to a group of K^2 sibling nodes is represented by $y_t = (x_{v_{1,1}}, \ldots, x_{v_{K,K}})$ where $v_{1,1}, \ldots, v_{K,K}$ share the same parent node u. Such tokenization allows representing the whole K^2–tree using $M(\log_{K^2}(N^2/M) + O(1))$ space, where N and M denote the number of nodes and edges in the original graph, respectively.

D. Generalizing K^2–tree to Attributed Graphs

Algorithm 3 Featured K^2–tree construction

Input: Modified adjacency matrix A and partitioning factor K.
Initialize the tree $\mathcal{T} \leftarrow (\mathcal{V}, \mathcal{E})$ with $\mathcal{V} = \emptyset, \mathcal{E} = \emptyset$. ▷ Featured K^2–tree.
Initialize an empty queue Q. ▷ Candidates to be expanded into child nodes.
Set $\mathcal{V} \leftarrow \mathcal{V} \cup \{r\}$, $x_r \leftarrow 1$ and let $A^{(r)} \leftarrow A$. Insert r into the queue Q. ▷ Add root node r.
while $Q \neq \emptyset$ **do**
    Pop u from $\mathcal{Q}$.
    if $x_u = 0$ **then** ▷ Condition for not expanding the node u.
        Go to line 4.
    end if
    Update $s \leftarrow \dim(A^{(u)})/K$
    for $i = 1, \ldots, K$ **do** ▷ Row-wise indices.
        for $j = 1, \ldots, K$ **do** ▷ Column-wise indices.
            Set $B_{i,j} \leftarrow A^{(u)}[(i-1)s : is, (j-1)s : js]$.
            ▷ Operation to obtain $s \times s$ submatrix $B_{i,j}$ of $A^{(u)}$.
            if $B_{i,j}$ is filled with zeros **then** ▷ Update tree-node attribute.
                Set $x_v \leftarrow 0$.
            else if $|B_{i,j}| > 1$ **then** ▷ Non-leaf tree-nodes with attribute 1.
                Set $x_v \leftarrow 1$.
            else ▷ Leaf tree-nodes with node features and edge features.
                Set $x_v \leftarrow B_{i,j}$. ▷ We treat 1×1 matrix $B_{i,j}$ as a scalar.
            end if
            if $\dim(B_{i,j}) > 1$ **then** $\mathcal{Q} \leftarrow v_{i,j}$.
            end if
        end for
    end for
    Set $\mathcal{V} \leftarrow \mathcal{V} \cup \{v_{1,1}, \ldots, v_{K,K}\}$. ▷ Update tree nodes.
    Set $\mathcal{E} \leftarrow \mathcal{E} \cup \{(u, v_{1,1}), \ldots, (u, v_{K,K})\}$. ▷ Update tree edges.
end while
Output: Featured K^2–tree $(\mathcal{T}, \mathcal{X})$ where $\mathcal{X} = \{x_u : u \in \mathcal{V}\}$.

Algorithm 4 Featured graph G construction

Input: Featured K^2–tree $(\mathcal{T}, \mathcal{X})$ and partitioning factor K.
$m \leftarrow K^{D\tau}$ ▷ Full adjacency matrix size.
Initialize $A \in \{0, 1\}^{m \times m}$ with zeros.
for $u \in \mathcal{L}$ **do** ▷ For each leaf node with $x_u \neq 0$.
    $\mathrm{pos}(u) = ((i_{v_1}, j_{v_1}), \ldots, (i_{v_L}, j_{v_L}))$. ▷ Position of node u.
    $(p_u, q_u) = (\sum_{\ell=1}^{L} K^{L-\ell}(i_{v_\ell} - 1) + 1, \sum_{\ell=1}^{L} K^{L-\ell}(j_{v_\ell} - 1) + 1)$. ▷ Location of node u.
    Set $A_{p_u, q_u} \leftarrow x_u$.
end for
Output: Modified adjacency matrix A.

In this section, we describe a detailed process to construct a K^2–tree for featured graphs with node features and edge features (e.g., molecular graphs), which is described briefly in Section 3.1. We modify the original adjacency matrix by incorporating categorical features into each element, thereby enabling the derivation of the featured K^2–tree from the modified adjacency matrix.

Edge features. Integrating edge features into the adjacency matrix is straightforward. It can be accomplished by simply replacing the ones with the appropriate categorical edge features.

Node features. Integrating node features into the adjacency matrix is more complex than that of edge features since the

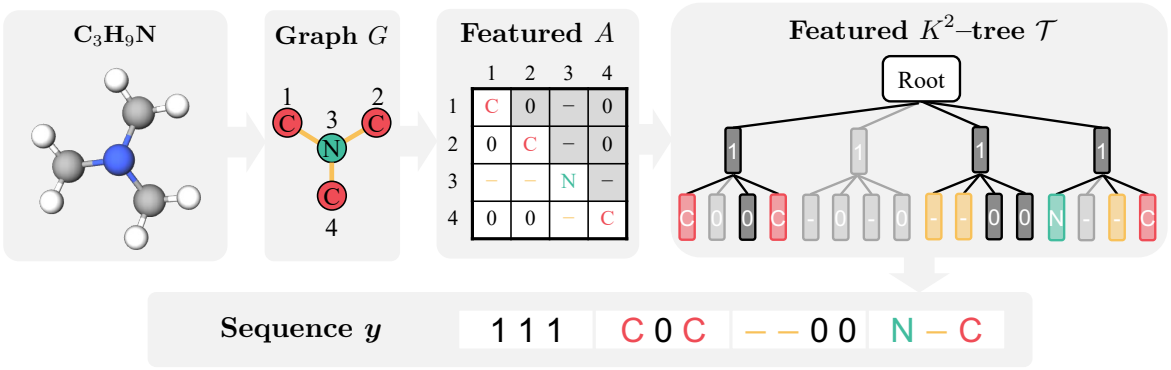


Figure 5. **An example of featured K^2–tree representation.** The shaded parts of the adjacency matrix and K^2–tree denote the redundant parts. The black-colored tree-nodes denote the normal tree-nodes with binary attributes while other-colored feature elements in the adjacency matrix A denote the same-colored featured tree-nodes and sequence elements. The node features (i.e., C and N) and edge feature (i.e., single bond $-$) of the molecule are represented within the leaf nodes.

adjacency matrix only describes the connectivity between node pairs. To address this issue, we assume that all graph nodes possess self-loops, which leads to filling ones to the diagonal elements. Then we replace ones on the diagonal with categorical node features that correspond to the respective node positions.

Let $x_u \in \mathcal{X}$ be the non-binary tree-node attributes that include node features and edge features and $\mathcal{L}$ be the set of leaf nodes in K^2–tree with non-zero node attributes. Then we can construct a featured K^2–tree with a modified adjacency matrix and construct a graph G from the featured K^2–tree as described in Algorithm 3 and Algorithm 4, respectively. In addition, we provide the illustration of the construction of K^2–tree in Appendix D.

E. Experimental Details

In this section, we provide the details of the experiments. Note that we chose $k = 2$ in all experiments and provide additional experimental results for $k = 3$ in Appendix H.

E.1. Generic graph generation

Table 3. **Hyperparameters of HGGT in generic graph generation.**

	Hyperparameter	Community-small	Planar	Enzymes	Grid
Transformer	Dim. of feedforward network	512	512	512	512
	Transformer dropout rate	0.1	0	0.1	0.1
	# of attention heads	8	8	8	8
	# of layers	3	3	3	3
Train	Batch size	128	32	32	8
	# of epochs	500	500	500	500
	Dim. of token embedding	512	512	512	512
	Gradient clipping norm	1	1	1	1
	Input dropout rate	0	0	0	0
	Learning rate	1×10^{-3}	1×10^{-3}	2×10^{-4}	5×10^{-4}

We used the same split with GDSS (Jo et al., 2022) for Community-small, Enzymes, and Grid datasets. Otherwise, we used the same split with SPECTRE (Luo et al., 2022) for the Planar dataset. We fix $k = 2$ and perform the hyperparameter search to choose the best learning rate in $\{0.0001, 0.0002, 0.0005, 0.001\}$ and the best dropout rate in $\{0, 0.1\}$. We select the model with the best MMD with the lowest average of three graph statistics: degree, clustering coefficient, and orbit count. Finally, we provide the hyperparameters used in the experiment in Table 5.

E.2. Molecular graph generation

Table 4. **Statstics of molecular datasets: QM9 and ZINC250k**.

Dataset	# of graphs	# of nodes	# of node types	# of edge types
QM9	133,885	$1 \leq \lvert V \rvert \leq 9$	4	3
ZINC250k	249,455	$6 \leq \lvert V \rvert \leq 38$	9	3

Table 5. **Hyperparameters of HGGT in molecular graph generation.**

	Hyperparameter	QM9	ZINC250k
Transformer	Dim. of feedforward network	512	512
	Transformer dropout rate	0.1	0.1
	# of attention heads	8	8
	# of layers	2	3
Train	Batch size	1024	256
	# of epochs	500	500
	Dim. of token embedding	512	512
	Gradient clipping norm	1	1
	Input dropout rate	0.5	0
	Learning rate	5×10^{-4}	5×10^{-4}

The statistics of training molecular graphs (i.e., QM9 and ZINC250k datasets) are summarized in Table 4 and we used the same split with GDSS (Jo et al., 2022) for a fair evaluation. We fix $k = 2$ and perform the hyperparameter search to choose the best number of layers in $\{2, 3\}$ and select the model with the best validity. In addition, we provide the hyperparameters used in the experiment in Table 5.

F. Implementation Details

F.1. Computing resources

We used PyTorch (Paszke et al., 2019) to implement HGGT and train the Transformer (Vaswani et al., 2017) models on GeForce RTX 3090 GPU.

F.2. Model architecture

We describe the architecture of the proposed transformer generator of HGGT in Figure 6. The generator takes a sequential representation of K^2–tree as input and generates the output probability of each token as described in Section 3.2. The model consists of a token embedding layer, transformer encoder(s), and multilayer perceptron layer with tree positional encoding.

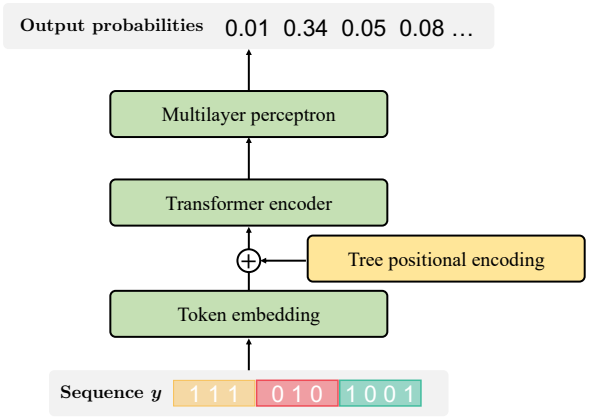


Figure 6. **The architecture of the transformer generator of HGGT.**

F.3. Details for baseline implementation

Generic graph generation. The baseline results from prior works are as follows. Results for GraphVAE (Simonovsky & Komodakis, 2018), GraphRNN (You et al., 2018), GNF (Liu et al., 2019), EDP-GNN (Niu et al., 2020), GraphAF (Shi et al., 2020), GraphDF (Luo et al., 2021), and GDSS (Jo et al., 2022) are obtained from GDSS, while the results for GRAN (Liao et al., 2019), SPECTRE (Martinkus et al., 2022), and GDSM (Luo et al., 2022) are derived from their respective paper. Additionally, we reproduced DiGress (Vignac et al., 2022) and GraphGen (Goyal et al., 2020) using their open-source codes. We used original hyperparameters when the original work provided them. DiGress takes more than three days for the Planar, Enzymes, and Grid datasets, so we report the results from fewer epochs after convergence.

Molecular graph generation. The baseline results from prior works are as follows. The results for EDP-GNN (Niu et al., 2020), MoFlow (Zang & Wang, 2020), GraphAF (Shi et al., 2020), GraphDF (Luo et al., 2021), GraphEBM (Liu et al., 2021), and GDSS (Jo et al., 2022) are from GDSS, and the GDSM (Luo et al., 2022) result is extracted from the corresponding paper. Moreover, we reproduced DiGress (Vignac et al., 2022) using their open-source codes.

G. Generated samples

In this section, we provide the visualizations of the generated graphs for generic and molecular graph generation.

G.1. Generic graph generation

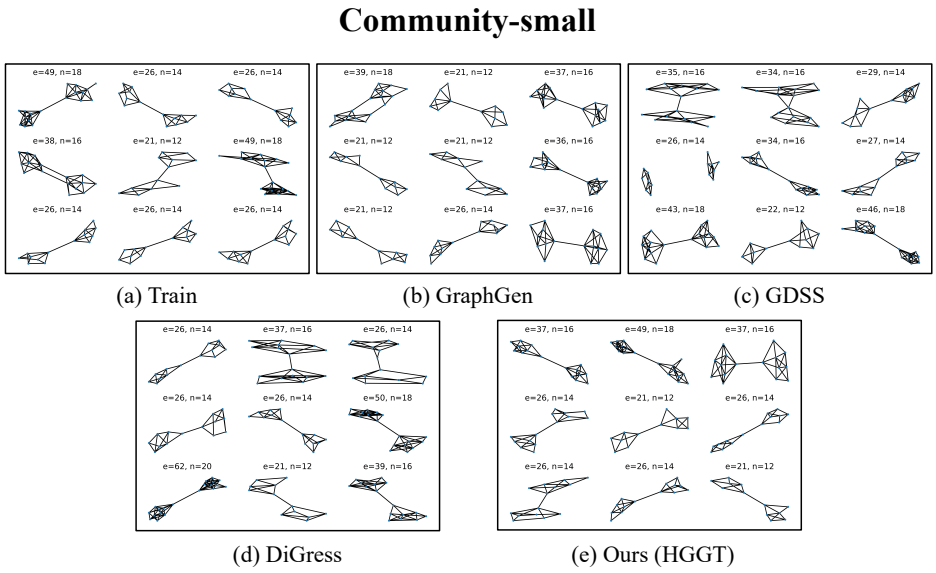


Figure 7. **Visualization of the graphs from the Community-small dataset and the generated graphs.**

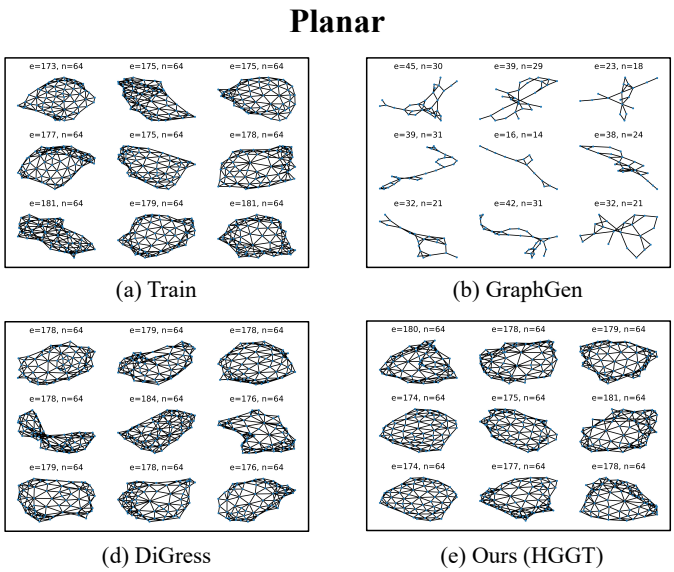


Figure 8. **Visualization of the graphs from the Planar dataset and the generated graphs.**

We present visualizations of graphs from the training dataset and generated samples from GraphGen, DiGress, GDSS, and HGGT in Figure 7, Figure 8, Figure 9, and Figure 10. Note that we reproduced GraphGen and DiGress using open-source codes while utilizing the provided checkpoints for GDSS. However, given that the checkpoints provided for GDSS do not include the Planar dataset, we have omitted GDSS samples for this dataset. We additionally give the number of nodes and edges of each graph.

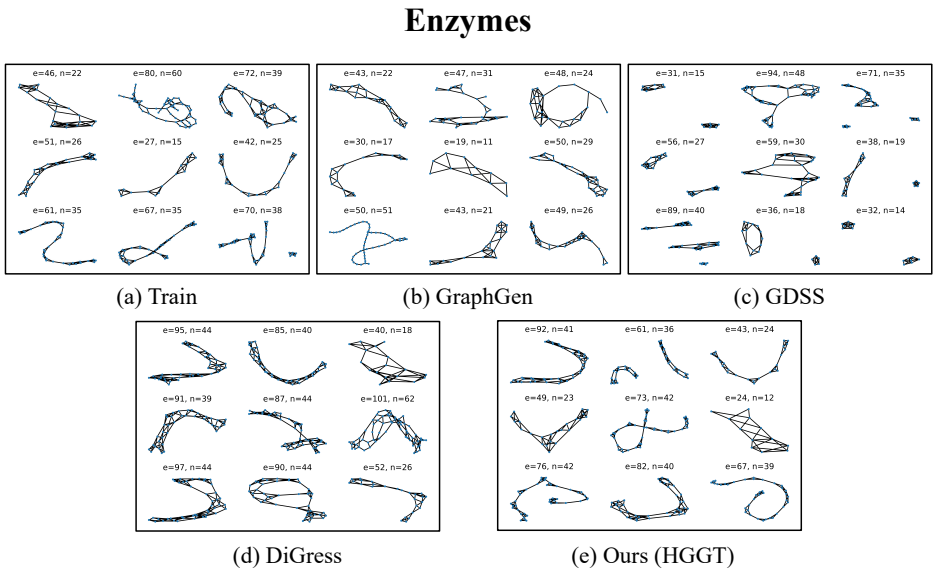


Figure 9. **Visualization of the graphs from the enzymes dataset and the generated graphs.**

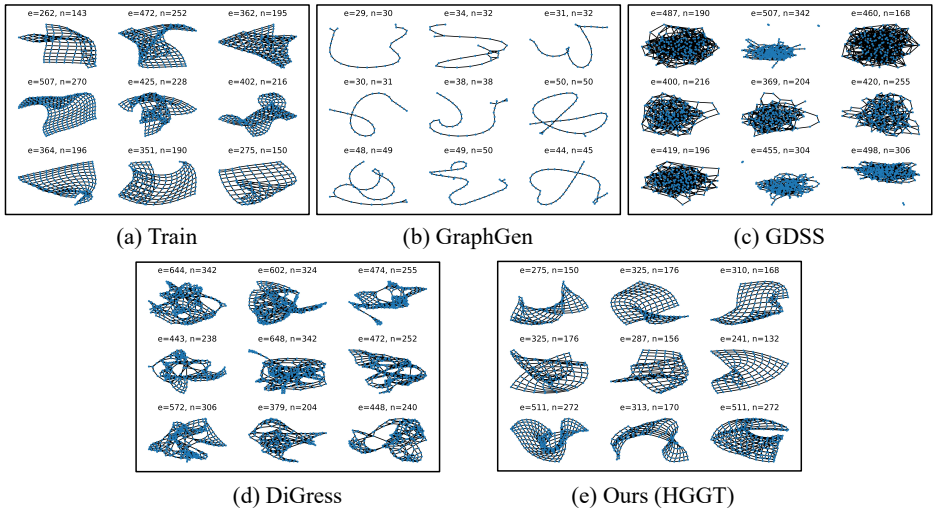


Figure 10. **Visualization of the graphs from the Grid dataset and the generated graphs.**

G.2. Molecular graph generation

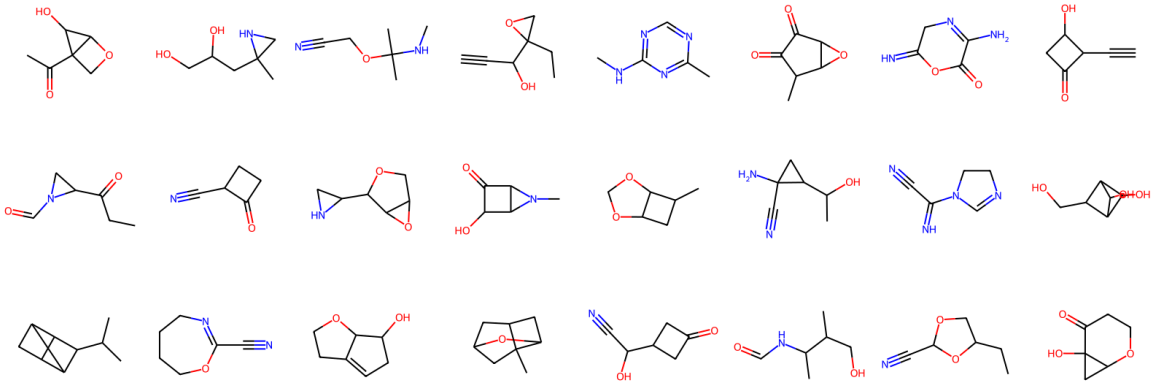


Figure 11. **Visualization of the molecules generated from the QM9 dataset.**

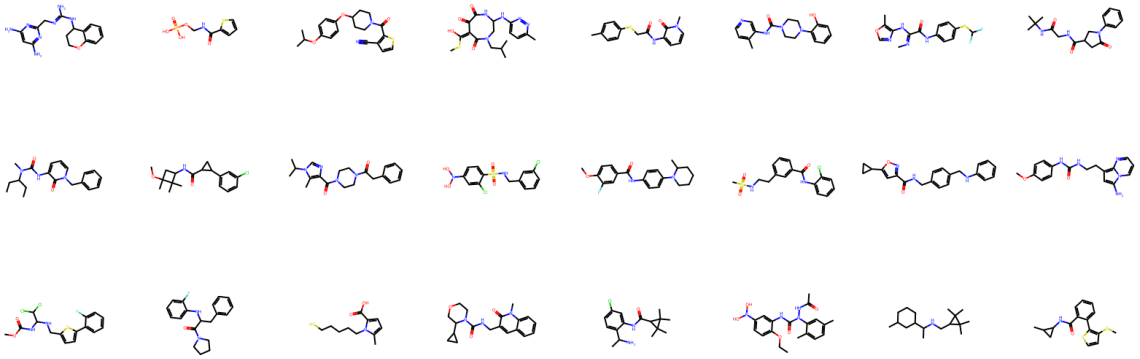


Figure 12. **Visualization of the molecules generated from the ZINC250k dataset.**

We present visualizations of generated molecules from HGGT in Figure 11 and Figure 12. Note that the 24 molecules are non-cherry-picked and randomly sampled.

H. Additional Experimental Results

In this section, we report additional experimental results.

H.1. Generic graph generation

We provide generic graph generation results for $k = 3$. Increasing k decreases the sequence length, while vocabulary size increases to $2^{3^2} + 2^6 = 578$.

We used Community-small and Planar datasets and measured MMD between the test graphs and generated graphs. We perform the same hyperparameter search for a fair evaluation as $k = 2$. The results are in Table 6. We can observe that HGGT still outperforms the baselines even with different k.

Table 6. **Generation results of HGGT with $k = 3$.**

	Community-small			Planar		
	Degree	Cluster.	Orbit	Degree	Cluster.	Orbit
$k = 2$	**0.001**	**0.006**	0.003	**0.000**	**0.001**	**0.000**
$k = 3$	0.007	0.050	**0.001**	0.001	0.003	**0.000**

H.2. Molecular graph generation

Table 7. **Additional molecular graph generation performance.**

	QM9								
Method	Frag. ↑	Intdiv. ↑	QED ↓	SA ↓	SNN ↑	Scaf. ↑	Weight ↓	Uniq. ↑	Nov. ↑
DiGress	0.9737	**0.9189**	0.0015	**0.0189**	**0.5216**	0.9063	**0.1746**	**96.34**	**35.46**
HGGT (Ours)	**0.9874**	0.9150	**0.0012**	0.0304	0.5156	**0.9368**	0.2430	95.65	24.01

	ZINC250k								
Method	Frag. ↑	Intdiv. ↑	QED ↓	SA ↓	SNN ↑	Scaf. ↑	Weight ↓	Uniq. ↑	Nov. ↑
DiGress	0.7702	**0.9061**	0.1284	1.9290	0.2491	0.0001	62.9923	**99.97**	**100**
HGGT (Ours)	**0.9877**	0.8644	**0.0164**	**0.2407**	**0.4383**	**0.5298**	**1.8592**	**99.97**	99.83

We additionally report nine metrics of the generated molecules: (a) fragment similarity (Frag.), which measures the BRICS fragment frequency similarity between generated molecules and test molecules, (b) internal diversity (Intdiv.), which measures the chemical diversity in generated molecules, (c) quantitative estimation of drug-likeness (QED), which measures the drug-likeness similarity between generated molecules and test molecules, (d) synthetic accessibility score (SA), which compares the synthetic accessibility between generated molecules and test molecules, (e) similarity to the nearest neighbor (SNN), an average of Tanimoto similarity between the fingerprint of a generated molecule and test molecule, (f) scaffold similarity (Scaf.), the Bemis-Murcko scaffold frequency similarity between generated molecules and test molecules, (g) weight, the atom weight similarity between generated molecules and test molecules, (h) the ratio of unique molecules (Uniq.), and (i) the ratio of novel molecules that did not appear in training molecules (Nov.). The results are in Table 7.

I. Ablation studies

Table 8. Ablation study for algorithmic components of HGGT.

			Community-small			Planar			Enzymes		
Group	PE	Prune	Degree	Cluster.	Orbit	Degree	Cluster.	Orbit	Degree	Cluster.	Orbit
✗	✗	✗	0.072	0.199	0.080	0.346	1.824	1.403	0.050	0.060	0.021
✓	✗	✗	0.009	0.105	**0.001**	0.003	**0.001**	0.002	0.005	0.022	0.007
✓	✓	✗	0.002	0.028	**0.001**	0.003	**0.001**	0.002	**0.002**	0.020	0.002
✓	✓	✓	**0.001**	**0.006**	0.003	**0.000**	**0.001**	**0.000**	0.005	**0.017**	**0.000**

We conduct three ablation studies to verify (1) the effectiveness of each component of sequential representation of K^2–tree, (2) the necessity of proper choice of positional encoding, and (3) the effect of node orderings on the compression ratio.

Table 9. Average graph compression ratio for different matrix orderings on generic graphs.

Method	Comm.	Planar	Enzymes	Grid
BFS	0.534	0.201	0.432	0.048
DFS	0.619	0.204	0.523	0.064
C-M	**0.508**	**0.195**	**0.404**	**0.045**

Ablation of algorithmic components. We introduced three components to enhance the performance of HGGT: grouping into tokens (Group), incorporating tree positional encoding (PE), and pruning the K^2–tree (Prune). To verify the effectiveness of each component, we present the experimental results for our method with incremental inclusion of these components. The experimental results are reported in Table 8. The results demonstrate the importance of each component in improving graph generation performance, with grouping being particularly crucial, thereby validating the significance of our additional components to the sequential K^2–tree representation.

Positional encoding. In this experiment, we assess the impact of various positional encodings in our method. We compare our tree positional encoding (TPE) to absolute positional encoding (APE) and relative positional encoding (RPE). Our findings, as presented in Figure 13, demonstrate that TPE outperforms other positional encodings with faster convergence of training loss. These observations highlight the importance of an appropriate choice of positional encoding for generating high-quality graphs.

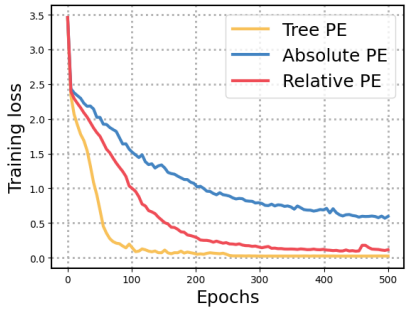


Figure 13. Training curve for different positional encodings on Planar.

Adjacency matrix orderings. It is clear that the choice of node order in the adjacency matrix A influences the size of K^2–tree. We verify our design choice to use of Cuthill-McKee (C-M) ordering (Cuthill & McKee, 1969) by comparing its compression ratio to two other common node orderings: breadth-first search (BFS) and depth-first search (DFS). The compression ratio is defined as the number of elements in K^2–tree divided by the number of cells in the adjacency matrix. In Table 9, we present the compression ratio results for each node ordering. Here, one can observe that C-M ordering shows the best compression ratio in most of the datasets compared to other orderings. Notably, K^2–tree demonstrates a good compression rate even after adding dummy nodes to construct a proper K^2–tree as claimed in Section 2.

J. Related Work

Adjacency matrix-based graph generation. The choice of graph representation is a crucial aspect of graph generation, as it can significantly impact the efficiency and expressiveness of the generative model. The most widely used representation is the adjacency matrix, which explicitly encodes the simple pairwise relationship between nodes (Simonovsky & Komodakis, 2018; Jo et al., 2022; Vignac et al., 2022; You et al., 2018; Liao et al., 2019; Niu et al., 2020; Shi et al., 2020; Luo et al., 2021). However, these methods suffer from high complexity in generating the adjacency matrix, especially for large graphs.

String-based graph generation. Researchers have developed string-based representations (Ahn et al., 2022; Krenn et al., 2019; Segler et al., 2018). In particular, for the molecular graph generation, Segler et al. (Segler et al., 2018) considered generating the SMILES representation (Weininger, 1988). Since the SMILES representation may be invalid, Krenn et al. (Krenn et al., 2019) designed a robust string-based representation of molecules. Finally, Ahn et al. (Ahn et al., 2022) designed a new representation to exploit the tree-like structure of molecules. For general graph generation, Goyal et al. (Goyal et al., 2020) designed a string representation using a graph canonicalization algorithm.

Motif-based graph generation. Researchers have investigated motif-based graph representations (Jin et al., 2018; 2020; Yang et al., 2021) that capture hierarchical graph structures with lower computational costs. Specifically, Jin et al. (Jin et al., 2018; 2020) considered extracting common fragments from data. Yang et al. (Yang et al., 2021) considered chemically reasonable fragments defined by domain experts. Finally, Guo et al. (Guo et al., 2022) considered learning motif-based grammar by running a reinforcement learning algorithm on the dataset. However, these methods rely on domain-specific knowledge to define or extract the motifs from data.

Lossless graph compression. Similar to the representations used for graph generation, lossless graph compression (Besta & Hoefler, 2018) aims to reduce the size and complexity of graphs while preserving their underlying structures. Specifically, several works (Brisaboa et al., 2009; Raghavan & Garcia-Molina, 2003) have introduced hierarchical graph compression methods that efficiently compress graphs by leveraging their hierarchical structure. In addition, Bouritsas et al. (Bouritsas et al., 2021) derived the compressed representation using a learning-based objective.