GSDiff: Synthesizing Vector Floorplans via Geometry-enhanced Structural Graph Generation - Supplements

Sizhe Hu, Wenming Wu*, Yuntao Wang, Benzhu Xu, Liping Zheng*

Hefei University of Technology

2024010072@mail.hfut.edu.cn, wwming@hfut.edu.cn, wyt@mail.hfut.edu.cn, {bzxu, zhenglp}@hfut.edu.cn

Preliminary

Diffusion models are a class of generative models that train neural networks to cleverly reverse a noise-adding process, enabling them to generate samples that simulate the original samples of the dataset from simple noise. A standard diffusion model typically consists of a forward process and a reverse process.

In the forward process, noise is gradually added to the original sample V_0 over T steps, resulting in a noisy sample V_T that resembles Gaussian noise. For a given time step t, this process generates a noisier version V_t using the following equation:

$$V_t = \sqrt{\overline{\alpha}_t} V_0 + \sqrt{1 - \overline{\alpha}_t}\epsilon \tag{1}$$

where $\overline{\alpha}_t = \alpha_1 \alpha_2 \cdots \alpha_t$ is the cumulative product of coefficients α_i used to control the proportion of the signal V_0 in V_t, and ϵ is standard Gaussian noise. As t increases, the proportion of noise becomes larger until it approaches the pure Gaussian noise.

The reverse process starts with Gaussian noise and gradually denoises it to reconstruct the original sample V_0. Less noisy versions of the original sample are recovered step by step using a neural network parameterized by θ, which outputs ϵ_θ as an estimation of the added noise. The probability distribution at time $t-1$ for the noisy sample V_t is given by

$$p_\theta(V_{t-1}|V_t) = \mathcal{N}(V_{t-1}; \mu_\theta(V_t, t), \sigma^2 I) \tag{2}$$

where

$$\mu_\theta(V_t, t) = \frac{1}{\sqrt{\alpha_t}} \left(V_t - \frac{1 - \alpha_t}{\sqrt{1 - \overline{\alpha}_t}} \epsilon_\theta(V_t, t) \right) \tag{3}$$

σ^2 is the variance associated with the diffusion process and $\mathcal{N}$ denotes the Gaussian distribution.

Node generation

Network architecture

The network architecture of the node Transformer (Figure 1) consists of stacked Transformer decoder layers (×L). The input to the node Transformer is the node set V_t and the time step t. For a node $v_i = (c_i, r_i, b_i) \in [-1, 1)^{2+7+1}$, we first obtain the node embedding $f_i \in \mathbb{R}^d$, where d denotes the embedding dimension. Specifically, for the positional coordinates of the node, we first denormalize $c_i = (x_i, y_i) \in [-1, 1)^2$ back to $[0, 255]^2$ and use the positional encoding proposed by (Vaswani et al. 2017), $f_i^c = [\gamma(x_i), \gamma(y_i)] \in \mathbb{R}^d$, where $\gamma(t)$ is defined as

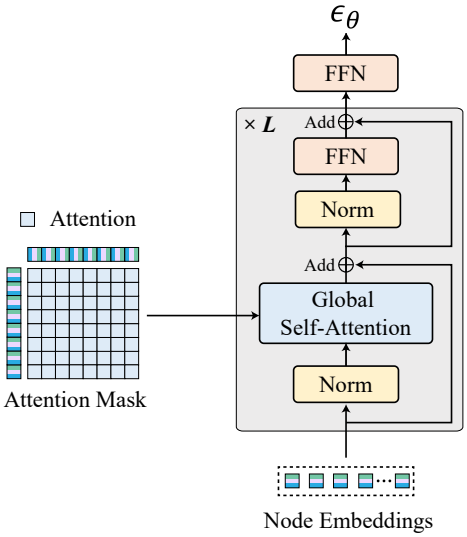

Figure 1: The network architecture of the node Transformer.

$$\gamma(t) = [\sin(\omega_0 t), \cos(\omega_0 t), \ldots, \sin(\omega_k t), \cos(\omega_k t)] \tag{4}$$

where $k = \frac{d}{4} - 1$, $\omega_j = \left(\frac{1}{10000}\right)^{\frac{4j}{d}}$, and d is the embedding dimension. For the semantic and background attributes (r_i, b_i) of the node, we use a fully connected layer to map (r_i, b_i) into a d-dimensional space to obtain the embedding $f_i^{(r,b)} \in \mathbb{R}^d$. The time step t is encoded to represent the noise level, and we use FFN to obtain the time embedding $f_i^t \in \mathbb{R}^d$. The final node embedding of v_i is obtained by fusing all three embeddings

$$f_i = f_i^c + f_i^{(r,b)} + f_i^t \tag{5}$$

The network architecture of the node Transformer consists of multiple decoder layers, with the input being the node

*Corresponding authors: Wenming Wu, Liping Zheng

embeddings of all nodes. Each decoder layer includes normalization, multi-head global self-attention, residual connections, and FFN, where the attention module models the dependencies between all nodes. Considering the variable number of nodes of each sample, we use background nodes to pad the node set to the same shape. The output of the decoder is a set of node embeddings, which, after passing through several decoder layers, are used to predict the noise parameters ϵ_θ of $p_\theta(V_{t-1}|V_t)$ at the next time step.

Alignment loss

High-quality structural graphs of architectural floorplans require geometric consistency, meaning nodes must align well. The alignment loss is used to measure the alignment error between nodes of the structural graph. Inspired by (Li et al. 2020), we define the alignment error of each node as the minimum distance between that node and all other nodes in any direction. Optimizing node alignment can be achieved by optimizing the sum of alignment errors of all nodes. We take the sum of alignment errors of all nodes as the alignment loss.

We use the predicted noise ϵ_θ and Equation 1 to predict node set $\hat{V}_0$. For $\hat{V}_0$, we have

$$\text{Alg}(\hat{V}_0) = \sum_{i=1}^{n} g(\min\left(\Delta c_i^X, \Delta c_i^Y\right)) \tag{6}$$

where n denotes the number of nodes in the node set, $g(x) = -d \cdot \log(1 - \frac{x}{d})$, $\Delta c_i^* = \min_{\forall j \neq i} |c_i^* - c_j^*|, * \in \mathcal{A} = \{X, Y\}$, and d represents the maximum allowable distance in direction $*$. Since the range of the positional coordinate for each node is $[-1, 1)$, $d = 1 - (-1) = 2$.

However, the inherent regression errors of neural networks present a challenge. Directly optimizing the regression loss, typically the MSE loss) of the neural network does not work well. One solution is to use a discrete coordinate representation to optimize the alignment loss. *HouseDiffusion* (Shabani, Hosseini, and Furukawa 2023) employs an 8-bit binary integer representation for coordinates within the range $[0, 255]$, which, in theory, could facilitate the model in learning precise alignment. However, this approach introduces notable risks in practice. For instance, consider a coordinate represented by the binary value $[1, 0, 0, 0, 0, 0, 1, 1]^2$ (131). If a prediction error occurs at the 4^{th} bit, the actual predicted result would be $[1, 0, 0, 0, 1, 0, 1, 1]^2$ (139), causing the coordinate to erroneously jump from 131 to 139. This magnitude of error significantly exceeds what would typically occur with direct regression methods.

To harmonize the advantages of binary representation and alignment loss, we convert the alignment error of each node into the binary form for regression. The discreteness introduced by the binary form helps to suppress noise to some extent, while the alignment loss constrains larger deviations. This method effectively avoids the pitfalls of high-bit inaccuracies and promotes coordinate alignment through discretization. The binary alignment loss is as follows

$$Alg^2(\hat{V}_0) = \sum_{i=1}^{n} g^2(\sum_{j=1}^{s} \left(Base^2\left(\min\left(\Delta c_i^X, \Delta c_i^Y\right)\right)\right)_j) \tag{7}$$

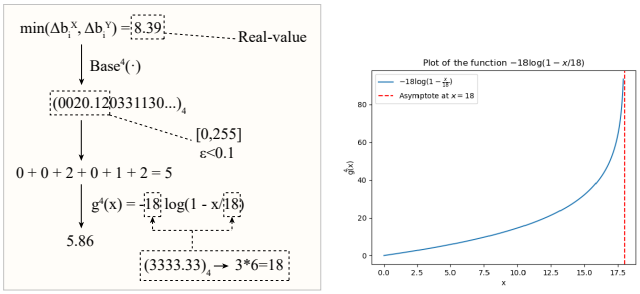

Figure 2: Quaternary alignment error. **Left**: The calculation process of a quaternary alignment error. **Right**: Plot of $g^4(x) = -18 \log\left(1 - \frac{x}{18}\right)$. $g^4(x)$ is used to impose penalties on larger alignment errors. When the independent variable (the alignment error of a single node) approaches 0, $g^4(x)$ approximates x; when the independent variable approaches $x = d^4 = 18$, an infinite penalty will be imposed. The larger the independent variable, the greater the penalty.

where $Base^2(\cdot)$ is the binary representation, $\Delta c_i^* = \min_{j \neq i} |c_i^* - c_j^*|, * \in \{X, Y\}$, $g^2(x) = -d^2 \log\left(1 - \frac{x}{d^2}\right)$, with d^2 indicating the maximum allowable distance under the binary representation. n is the node number, s is the bit size. To calculate the binary alignment loss, we convert the coordinate range from $[-1, 1)$ to $[0, 255]$ (corresponding to 8 bits). We take a precision of 1×10^{-1} (corresponding to 4-bit binary fraction), so s is set to 12 in our experiments. We use the L1 norm of the binary vector to represent the binary distance, therefore,

$$d^2 = \left\| \left[\underbrace{1, 1, 1, \ldots, 1, 1, 1}_{12}\right]^2 - \left[\underbrace{0, 0, 0, \ldots, 0, 0, 0}_{12}\right]^2 \right\|_1 = 12 \tag{8}$$

As we convert the coordinate range from $[-1, 1)$ to $[0, 255]$, we finally normalize the binary alignment loss by a scaling factor of $1/128$. The notation $(\cdot)_j$ represents the j-th bit.

Binary learning treats high and low bits equally important, which is not optimal for gradient-based optimization. To balance precision and optimizability, we propose a mixed-base optimization strategy, combining multiple numeric bases: we trade off computational complexity (which increases linearly with the number of bases) and representation granularity by adding quaternary, octal, and hexadecimal bases to the binary and real number representations. The quaternary, octal, and hexadecimal losses are similar to those in binary. The bit size s is 6, 5, and 3 for these three bases, respectively. The L1 norm of the corresponding base vectors d^k is 18, 35, and 45, respectively. The normalization scaling factor is also $1/128$. Without loss of generality, we illustrate the calculation process of a quaternary alignment error in Figure 2.

The final mixed-base alignment loss is given by

$$MixAlg(\hat{V}_0) = Alg(\hat{V}_0) + \sum_{k \in 2,4,8,16} Alg^k(\hat{V}_0) \tag{9}$$

LACE (Chen et al. 2024) suggests applying alignment loss at larger time t, which may degrade performance, so they use a

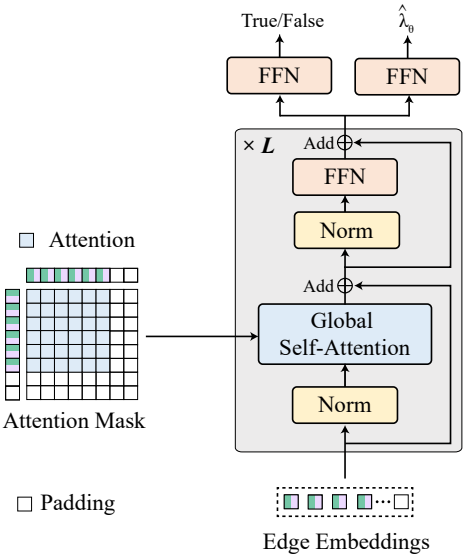

Figure 3: The network architecture of the edge Transformer.

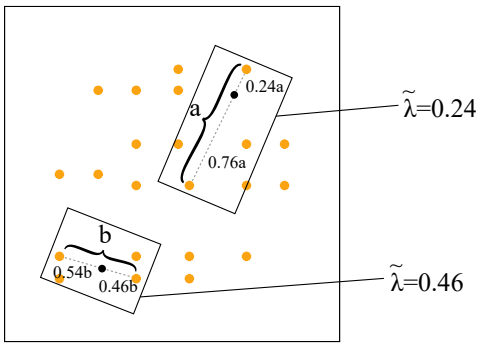

Figure 4: Edge perception enhancement. This figure shows the relationship between $\tilde{\lambda}$ and the position of the interpolation point for candidate edges of lengths a and b. For the candidate edge of length a, the interpolation point divides the candidate edge into two segments with a length ratio of 0.76:0.24 (regardless of whether the interpolation coefficient λ is 0.24 or 0.76), then $\tilde{\lambda}$ is equal to the smaller of the two, which is 0.24. The interpolation coefficient λ is random each time, ensuring that every point on the entire candidate edge can be selected.

time-dependent constraint weight, applying larger alignment loss weights only at smaller time t. We also apply this weight function, multiplying it with the alignment loss: $\omega(t) = 1 - \alpha_{T-t}$ where α_{T-t} is from Equation 1.

Clamping

During sampling (i.e., the reverse process of the diffusion model), the neural network's estimation of data at $t = 0$ may exceed reasonable ranges, causing deviation in the sampling path and degrading generation quality. The clamping technique limits the neural network's output to reasonable ranges to prevent such occurrences. In our case, the normalized coordinates for node attributes are limited to $[-1, 1)^2$, and other attributes are constrained to $\{0, 1\}$. According to Equation 2, the predicted noise ϵ_θ in the reverse process will be converted into the prediction $\hat{V}_0$. We clamp $\hat{V}_0$ within these ranges at each step of the reverse process to improve generation quality. We set the threshold of all semantic attributes to 0.5 and the threshold of background attributes to 0.75 to avoid missing nodes.

Edge prediction

Network architecture

We use the edge Transformer to perform binary classification ("True/False") on the candidate edges. The network architecture of the edge Transformer (Figure 3) consists of stacked Transformer decoder layers (×L), each comprising normalization, multi-head global self-attention, residual connections, and FFN. The input to the Edge Transformer consists of the embeddings of the candidate edge set E' and the padding mask constructed for E', where an edge is considered non-padding only if both of its endpoints are non-padding. After passing through several decoder layers, the output embeddings of the candidate edges are fed through an FFN to predict the authenticity of the edges. All edges identified as "True" form the edge set E, which is the output of the edge prediction.

Edge perception enhancement

In general, the edge prediction model only uses the features of two endpoints as the edge features, which lack sufficient geometric information, resulting in missing or false edges and making the edge prediction unreasonable. To enhance the ability of edge perception, we propose a self-supervised edge perception enhancement strategy, as shown in Figure 4. Specifically, for each candidate edge, we introduce a third point at a random location on the edge, determined by a random coefficient λ. As a result, each edge is no longer represented solely by two endpoints, but by two endpoints along with a randomly interpolated point. The model is trained to predict this interpolation coefficient λ, which forces the model to better perceive edges. This strategy improves the geometric reasoning capability of the model by introducing additional geometric features for each edge.

For each candidate edge (v_i, v_j), the enhanced edge features include the features of both endpoints, as well as the feature combination of the random interpolation point. The interpolation point feature is determined by the interpolation coefficient λ, which decides the location of the interpolation point on the edge. The coordinates and semantics of the interpolation point are as follows

$$c_\lambda = \lambda c_i + (1 - \lambda) c_j \in [-1, 1)^2 \tag{10}$$

$$r_\lambda = \mathbf{0} \in \mathbb{R}^7 \tag{11}$$

where $\lambda \sim U(0, 1)$ is a uniformly distributed interpolation coefficient sampled from the range $[0, 1]$. The zero vector $\mathbf{0}$ represents semantics, as the semantics at the random interpolation point on any candidate edge do not contribute to

geometric reasoning. The loss for the self-supervised term is defined as

$$\tilde{\lambda} = \begin{cases} 1-\lambda, & \text{if } \lambda > 0.5, \\ \lambda, & \text{otherwise.} \end{cases} \tag{12}$$

$$\mathcal{L}_{\lambda} = \mathbb{E}\left[\left|\tilde{\lambda} - \hat{\lambda}_{\theta}(e_{ij})\right|\right] \tag{13}$$

where $\hat{\lambda}_{\theta}(e_{ij})$ represents the predicted interpolation coefficient. We process the original interpolation coefficient $\lambda \in [0, 1]$ as follows: if $\lambda > 0.5$, we use $1 - \lambda$; otherwise, we keep it as is. This is done because the edges are undirected, so the model cannot distinguish which endpoint is the reference point during prediction. The total loss is defined as

$$\mathcal{L}_{\text{edge}} = \mathcal{L}_{\text{cls}} + \mathcal{L}_{\lambda} \tag{14}$$

where $\mathcal{L}_{cls}$ is the Cross-entropy classification loss, and $\mathcal{L}_{\lambda}$ is the interpolation coefficient regression loss.

It is noteworthy that our edge perception enhancement strategy differs significantly from the mixup technique (Zhang et al. 2017) in both implementation and objective. The mixup technique creates new samples by blending two random samples from the training data, aiming to improve the model's generalization and its resistance to adversarial samples. In contrast, our strategy is tailored to enhance the model's perception of the geometric characteristics of edges within a structure. We assign a distinct interpolation coefficient to each candidate edge and train the model to predict this coefficient, enabling the model to discern the continuity of points along the edge, treating the candidate edge as a whole line segment rather than just two endpoints. This is crucial for the edge prediction task, which involves geometric inference. Unlike the mixup technique, whose goal is to smooth the decision boundary in the model's output space, our strategy focuses on enriching the input feature space with geometric features that are directly pertinent to the current task.

Floorplan extraction

The floorplan extraction process is easy to implement. We can simply extract all minimal polygonal cycles as rooms. This process involves starting from a certain edge in the structural graph, following a predefined order of nodes, moving from the smaller-numbered endpoint to the larger-numbered endpoint, and consistently turning the maximum angle in a fixed clockwise (or counterclockwise) direction to select the corresponding next edge. This process continues until it returns to the starting edge, and all edges traversed during this process form a minimal polygonal cycle. This procedure is illustrated in Figure 5. Specifically, we traverse all edges to obtain all polygonal cycles and select the non-duplicate ones. In practice, we apply the following simple rule to avoid obtaining duplicate polygons: for each edge, if it has been traversed in the order from the smaller-numbered endpoint to the larger-numbered endpoint, it is marked as visited and will not be visited again. Each time, we only traverse edges that have not been visited.

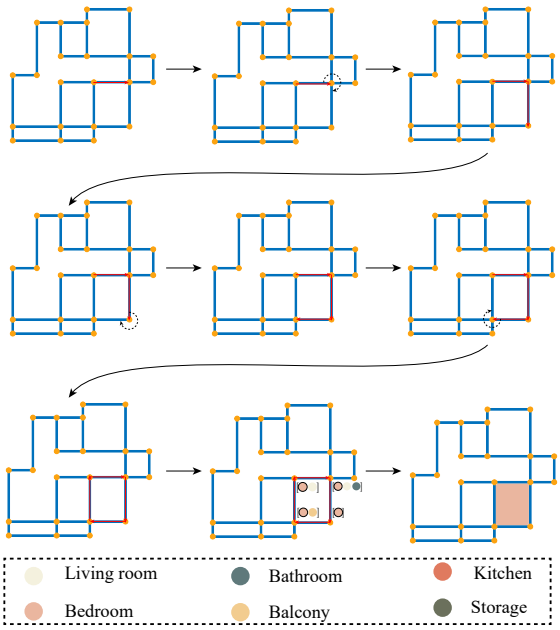

Figure 5: An example of floorplan extraction.

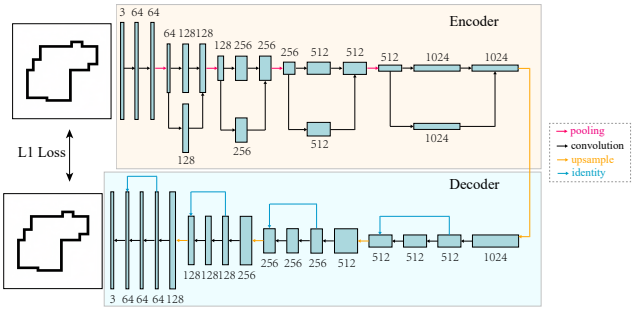

Figure 6: The network architecture of the autoencoder for boundary constraints.

Constrained generation

Training paradigm

We do not directly train the constraint encoder and node generation model jointly, as this would limit training on a training dataset of finite size. Given the critical importance of the quality of the constraint encoder, we first train each constraint encoder separately to achieve near-lossless performance, and then train the constraint encoder and node generation model jointly. To train the constraint encoder, we adopt a pre-training + fine-tuning paradigm: we first perform pre-training using randomly generated samples (which can be considered as having an infinite size of the training dataset), and then fine-tune the model on our training dataset. This strategy enhances generalization capability. The training is based on the reconstruction loss of the autoencoder.

Boundary-constrained generation

In architectural floorplans, the boundary refers to the contour formed by the outer walls of the building, typically represented as a polygon. Given its geometric nature, we use the Convolutional Neural Network (CNN) to encode it. Specifically, during the pre-training phase, the input to the CNN is heuristically constructed. We determine each polygon vertex through a random walk on a 256×256 three-channel blank image, where each vertex's 2D coordinates are uniformly sampled across the entire image. The edges of the polygon are drawn in layers with different colors: green (7 pixels), blue (5 pixels), red (3 pixels), and black (1 pixel). The number of vertices, corresponding to the number of random walk steps, is sampled from the training dataset.

We modified the U-Net structure (Ronneberger, Fischer, and Brox 2015) by removing the skip connections and converting it into an autoencoder, adding residual connections between layers to improve performance. As the network depth increases, the number of channels in the encoder gradually increases, and we replace identity mapping with 1×1 convolution. The network output is also a 256×256 three-channel image, and we compute the L1 loss between the input and output images. Figure 6 illustrates the network architecture of the autoencoder for boundary constraints. For fine-tuning, we use the real boundary sample from the training dataset. The boundaries are drawn with black lines 7 pixels wide, and the same loss is used for training.

Topology-constrained generation

The topological graph is defined as an undirected graph $G_{\text{top}} = (V_{\text{top}}, E_{\text{top}})$, where each vertex $v_{\text{top},i} = (r_i) \in V_{\text{top}}$ represents a room, and a one-hot encoded vector $r_i \in \{0,1\}^7$ indicates the room category. The edge $(v_{\text{top},i}, v_{\text{top},j}) \in E_{\text{top}}$ represents an adjacency relationship between a pair of rooms (i.e., whether they share a wall). Considering the graph-like nature of this problem, we use the topology Transformer to encode it.

In the pre-training, we pre-compute room counts (ranging from 4 to 8), adjacency relationships ("True" or "False"), and room categories from the dataset. We then randomly sample the room count, adjacency relationships, and room categories to construct random topological graphs, which are used as input to the topology Transformer. Since the room count in the topological graph is uncertain, padding is required, and a cross-attention mask is used to limit attention to all nodes and real rooms. The input to the topology Transformer is G_{top} and the cross-attention mask, where the room set V_{top} attends to the given adjacency relationships E_{top}. After passing through several encoder layers, the output is the room embeddings.

In the fine-tuning, we perform room classification and adjacency relationship classification tasks on the topological graph as reconstruction tasks: room classification forces room embeddings to contain their category information, while adjacency relationship classification forces room embeddings to contain the correct adjacency relationships. As a result, room embeddings effectively encode the entire information of the topological graph, serving as a constraint for node generation. Both room classification and adjacency relationship classification use the cross-entropy loss. The network architecture of the topology Transformer is shown in Figure 7. We fine-tune using the real topological graphs from the training dataset.

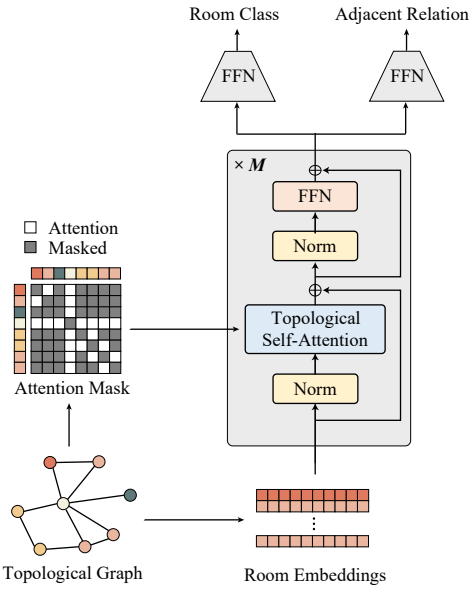

Figure 7: The network architecture of the topology Transformer. Gray in attention mask: which attention are masked.

Experimental setup

The implementation details of the different networks are as follows, with hyperparameters chosen based on empirical observations and performance on the validation dataset.

Unconstrained generation

Node Transformer The node Transformer consists of 24 layers with an embedding dimension of 256, resulting in a total of 19 million parameters. The batch size is set to 256, and the optimizer is Adam (Kingma 2014). The training is conducted for 1,000,000 steps. The initial learning rate is 1×10^{-4}, which is reduced by a factor of 0.1 after 500,000 steps.

Edge Transformer The edge Transformer consists of 12 layers with an embedding dimension of 256, resulting in a total of 10 million parameters. The batch size is set to 8. Given the significant impact of edge prediction quality on the results, we implement a learning rate decay strategy for training: the initial learning rate is 1×10^{-4}, and the optimizer is Adam (Kingma 2014). The performance on the validation dataset is closely monitored, and if the validation metric shows no improvement for five consecutive evaluations, the learning rate is reduced by a factor of 0.1. If there is no improvement for 20 consecutive evaluations, training is terminated, and the model with the best performance on the validation dataset is selected. Evaluation is performed every

1,000 steps. The performance of the edge prediction model peaked at 61,000 steps.

Constrained generation

For the constrained generation, we increased the embedding dimension of the node Transformer to 512, resulting in a total of 96 million parameters. This change is made as we have observed that using the same 19 million parameters, adding topological constraints could reduce performance, possibly due to the difficulty of accommodating multiple types of information such as topological graphs and coordinates within the 256-dimensional space. The boundary constraints are configured similarly. The configuration of the Edge Transformer remained unchanged.

Boundary CNN The boundary CNN has 31 million parameters. During the pre-training phase, the batch size is set to 16, and the optimizer is Adam (Kingma 2014) with an initial learning rate of 1×10^{-4}. We use the same learning rate decay strategy for training as the edge Transformer. The performance peaked at 5,000 steps during pre-training. For fine-tuning, we restarted the learning rate at 1×10^{-4} with a batch size of 16, and evaluation was performed every 100 steps, continuing training until 6,700 steps.

Topology Transformer The topology Transformer consists of 24 layers with an embedding dimension of 256, resulting in a total of 19 million parameters. During the pre-training phase, the batch size is set to 2048, and the optimizer is Adam (Kingma 2014) with an initial learning rate of 1×10^{-4}. We use the same learning rate decay strategy for training as the edge Transformer. The performance peaked at 11,000 steps during pre-training. For fine-tuning, we restarted the learning rate at 1×10^{-4} with a batch size of 256, continuing training for 6,000 steps with the same method.

Dataset

We first extract the 2D coordinates of wall junctions and segments. The image is binarized, with the walls represented as white and all other areas as black. Due to varying wall thicknesses, we regularize the thickness through repeated morphological operations (erosion, dilation) and template matching. We iteratively erode the white pixels representing the wall junction components until the next erosion step results in a decrease in the number of connected components in the image. This indicates that some wall segments have been eroded to a thickness of 1 pixel.

Next, we apply a series of template matching operations, sliding a 3×3 window across the image at this stage. If a match is successful, the matching area (the 3×3 window) is marked as white. The templates represent lines with a thickness of 1 pixel or local wall shapes that are defective. Through this process, the shape of the walls is gradually standardized, and the thickness becomes more uniform. This iterative process continues until the erosion reduces the thickness of all walls to 1 pixel; at this point, further erosion would cause all walls to disappear, turning the image completely black. At this stage, the wall structure is what we require.

Finally, the wall junctions and segments are extracted from the image, where the wall thickness has been uniformly reduced to 1 pixel. We obtain semantics from the four-channel images of the original RPLAN dataset. Any images that fail to process at this step are discarded. We obtain 71,763 vectorized floorplan images, randomly splitting them into 3,000 for the validation set, 3,000 for the test set, and the remainder for the training set. In the original RPLAN dataset, rooms are divided into 14 categories, which we merged into 7 categories: *Living room*, *Bedroom*, *Kitchen*, *Bathroom*, *Balcony*, *Storage*, *External area*.

Figure 8 (b1) shows some floorplan samples obtained from the above process. The RPLAN dataset (Wu et al. 2019) comes from real residential layouts, which do not contain slanted walls. To verify that our method is also applicable to floorplans with slanted walls, we heuristically deform the "peninsula-like" rectangular balcony which surrounded by the *External area* on three sides into an isosceles trapezoid with the top base being 0.618 times the length of the bottom base. The data after the slanting deformation is shown in Figure 8 (b2).

We have conducted distributional statistics (Figure 8(a)) on the processed RPLAN dataset (Wu et al. 2019), including the number of wall junctions, the number of wall segments, the number of rooms, and the quantity of each room category.

More results

Figure 9 displays a comparison of boundary-constrained generation across different methods, including the ground-truth (GT), *Graph2Plan* (Hu et al. 2020), *WallPlan* (Sun et al. 2022), and ours. Figure 10 provides a comparison of topology-constrained generation among various techniques. including *HouseDiffsion* (Shabani, Hosseini, and Furukawa 2023), *House-GAN++* (Nauata et al. 2021) and ours. Figure 11 showcases the results of unconstrained generation. Figure 12 showcases the results of unconstrained generation with slanted walls. Figures 13 and 14 present the results of boundary-constrained generation by our method, while Figures 15 and 16 illustrate the results of topology-constrained generation by our method.

References

Chen, J.; Zhang, R.; Zhou, Y.; and Chen, C. 2024. Towards Aligned Layout Generation via Diffusion Model with Aesthetic Constraints. *arXiv preprint arXiv:2402.04754*.

Hu, R.; Huang, Z.; Tang, Y.; Van Kaick, O.; Zhang, H.; and Huang, H. 2020. Graph2plan: Learning floorplan generation from layout graphs. *ACM Transactions on Graphics (TOG)*, 39(4): 118–1.

Kingma, D. 2014. Adam: a method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

Li, J.; Yang, J.; Zhang, J.; Liu, C.; Wang, C.; and Xu, T. 2020. Attribute-conditioned layout gan for automatic graphic design. *IEEE Transactions on Visualization and Computer Graphics*, 27(10): 4039–4048.

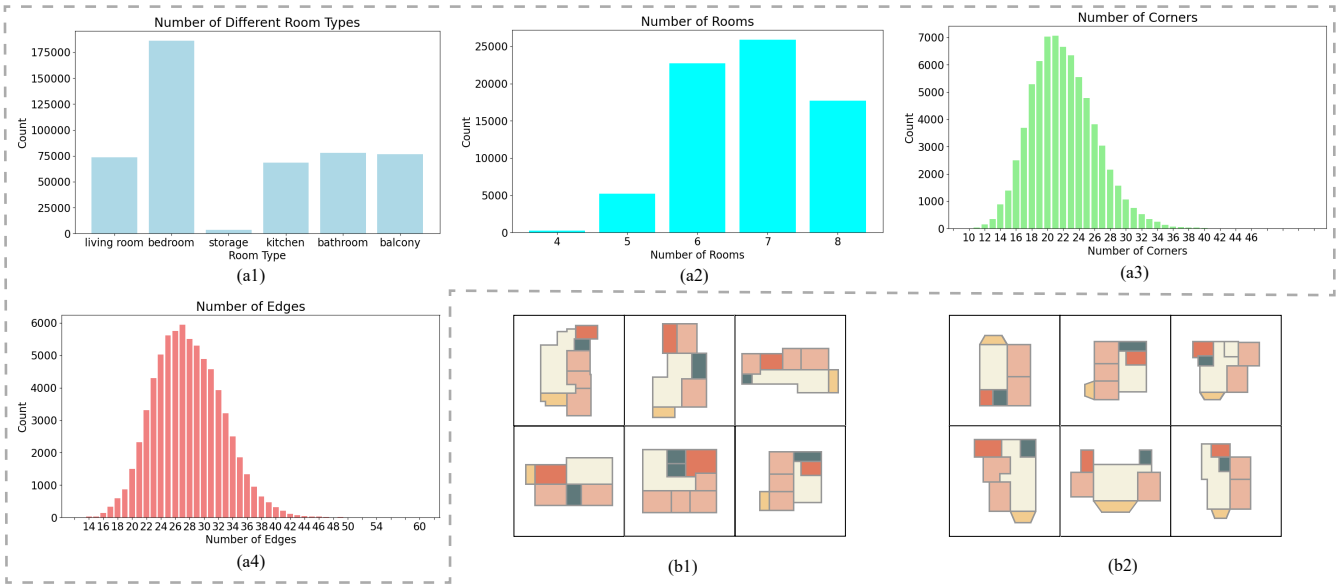

Figure 8: Dataset. (a1)-(a4): distributional statistics on the processed RPLAN dataset (Wu et al. 2019), including the quantity of each room category (a1), the number of rooms (a2), the number of wall junctions (a3) and the number of wall segments (a4). (b1)-(b2): some data samples obtained from the above data processing, (b1) shows original floorplans, and (b2) shows floorplans with slanted walls.

Nauata, N.; Hosseini, S.; Chang, K.-H.; Chu, H.; Cheng, C.-Y.; and Furukawa, Y. 2021. House-gan++: Generative adversarial layout refinement network towards intelligent computational agent for professional architects. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 13632–13641.

Ronneberger, O.; Fischer, P.; and Brox, T. 2015. U-net: Convolutional networks for biomedical image segmentation. In *Medical image computing and computer-assisted intervention–MICCAI 2015: 18th international conference, Munich, Germany, October 5-9, 2015, proceedings, part III 18*, 234–241. Springer.

Shabani, M. A.; Hosseini, S.; and Furukawa, Y. 2023. Housediffusion: Vector floorplan generation via a diffusion model with discrete and continuous denoising. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 5466–5475.

Sun, J.; Wu, W.; Liu, L.; Min, W.; Zhang, G.; and Zheng, L. 2022. Wallplan: synthesizing floorplans by learning to generate wall graphs. *ACM Transactions on Graphics (TOG)*, 41(4): 1–14.

Vaswani, A.; Shazeer, N.; Parmar, N.; Uszkoreit, J.; Jones, L.; Gomez, A. N.; Kaiser, L. u.; and Polosukhin, I. 2017. Attention is All you Need. In Guyon, I.; Luxburg, U. V.; Bengio, S.; Wallach, H.; Fergus, R.; Vishwanathan, S.; and Garnett, R., eds., *Advances in Neural Information Processing Systems*, volume 30. Curran Associates, Inc.

Wu, W.; Fu, X.-M.; Tang, R.; Wang, Y.; Qi, Y.-H.; and Liu, L. 2019. Data-driven interior plan generation for residential buildings. *ACM Transactions on Graphics (TOG)*, 38(6): 1–12.

Zhang, H.; Cisse, M.; Dauphin, Y.; and Lopez-Paz, D. 2017. mixup: Beyond Empirical Risk Minimization.

Figure 9: More on the comparison of boundary-constrained generation across different methods.

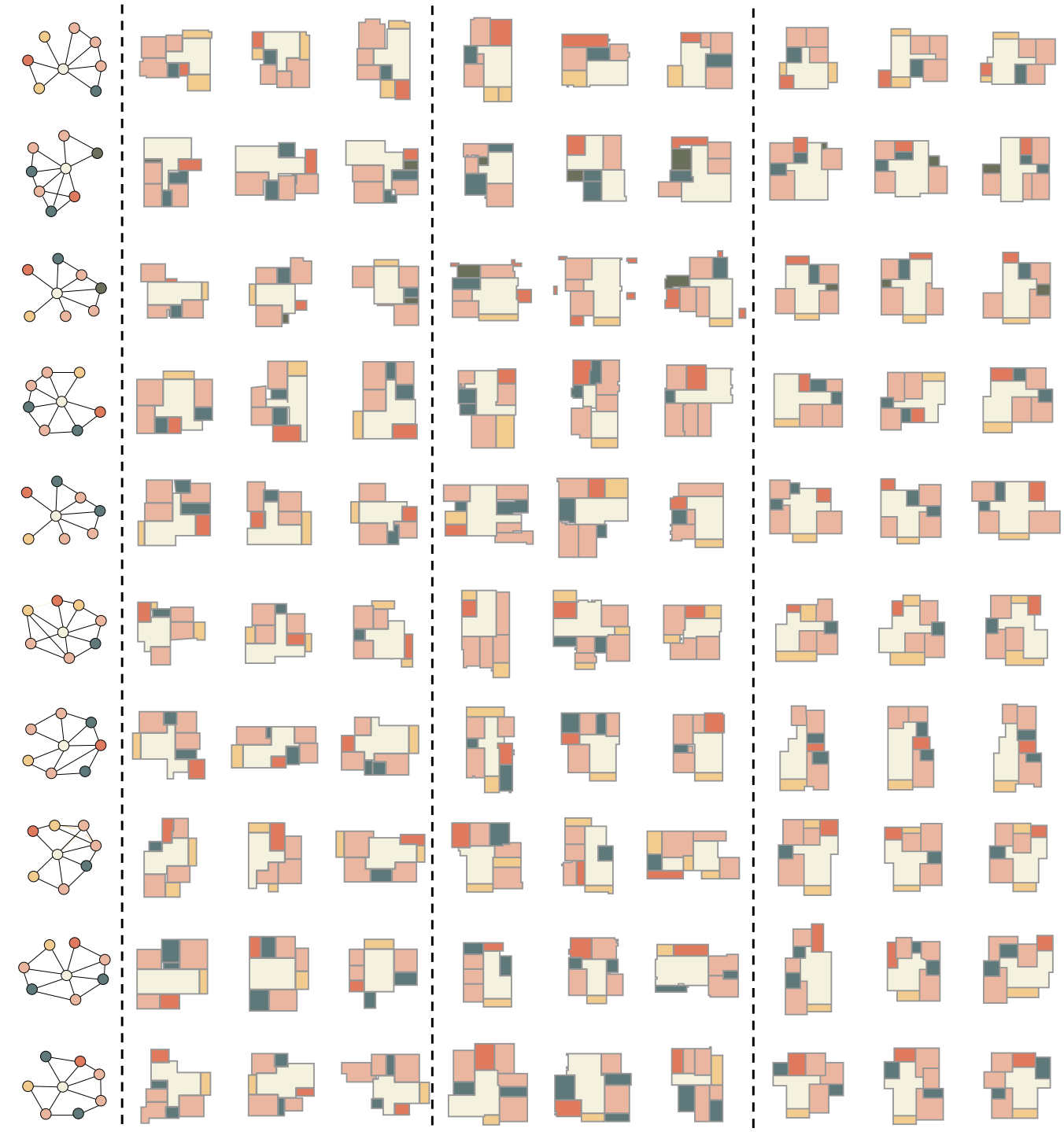

Figure 10: More on the comparison of topology-constrained generation among various techniques.

Figure 11: More results of unconstrained generation by our method.

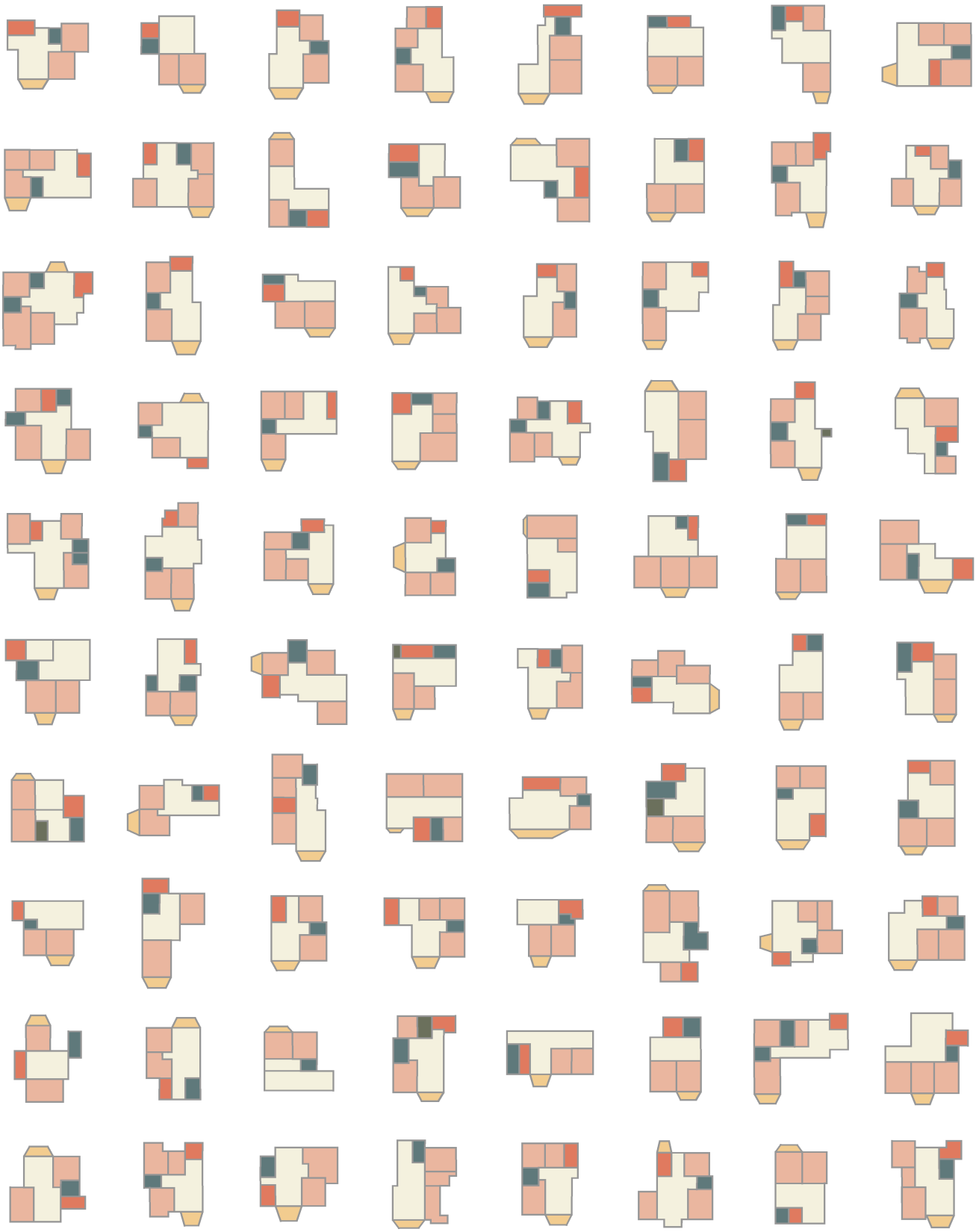

Figure 12: More results of unconstrained generation with slanted walls by our method.

Figure 13: More results of boundary-constrained generation by our method: part (I).

Figure 14: More results of boundary-constrained generation by our method: part (II).

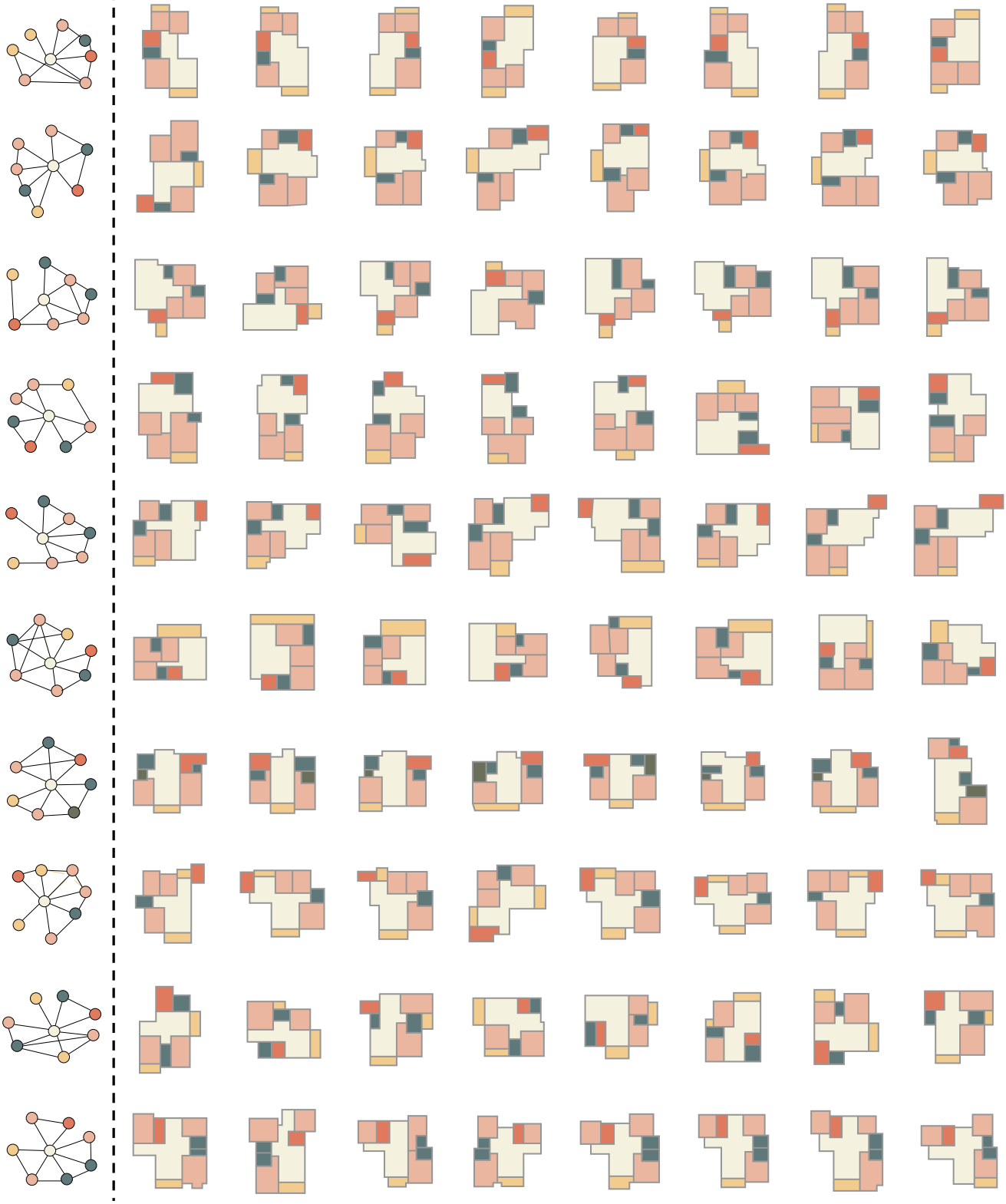

Figure 15: More results of topology-constrained generation by our method: part (I).

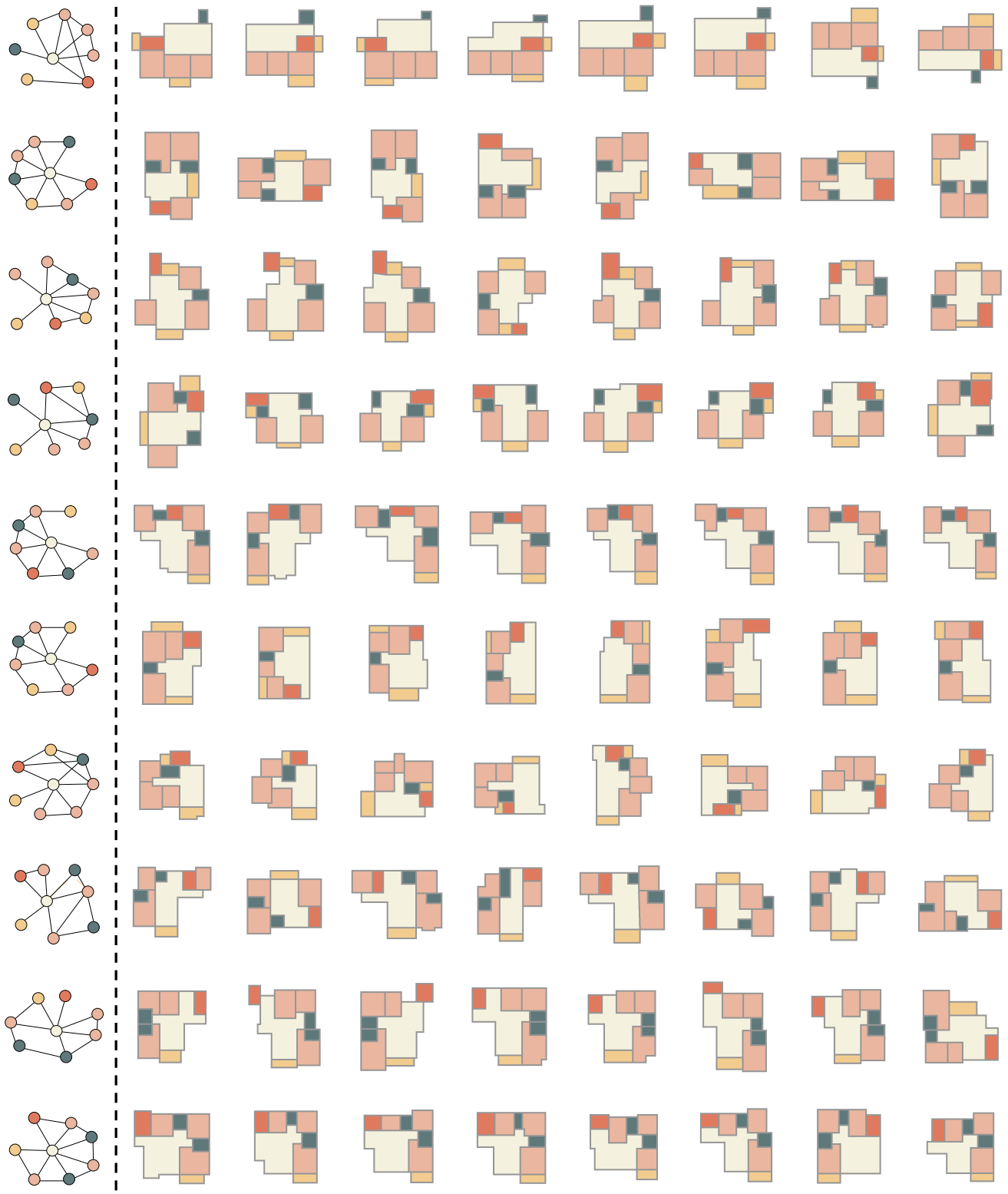

Figure 16: More results of topology-constrained generation by our method: part (II).