LES ACCROGRAPHES
Rosalie MILLNER
Emilio PICARD
rosalie@millner.fr
emilio.picard@free.fr

Graph Generation with Specified Properties

ALTEGRAD 2024

01/15/25
MVA

1 Introduction

In recent years, there has been an increasing interest in generating graphs given specified properties, being an area that finds applications ranging from chemo-informatics to social networks and infrastructure modeling. The complexity and high-dimensional nature of graph properties make graph generation a particularly challenging problem in machine learning. This project, part of a Kaggle competition, explores conditional graph generation using latent diffusion models. By conditioning the generation process on graph properties extracted from textual descriptions, the goal is to produce graphs that accurately match with the specified characteristics. We present in this report the methodology, experimentation, and insights we developed throughout the project.

2 Background

The baseline model for the project is based on the NGG model given in [1], which combines a Variational Graph Autoencoder with a conditioned latent diffusion process. These are the main components:

- **Variational Graph Autoencoder (VGAE)**: This module compresses graphs into a lower-dimensional latent space. For a given graph G, the encoder transforms it into a latent vector $z = E(G)$, and the decoder uses this representation to reconstruct the original graph $G = D(z) = D(E(G))$. This latent representation acts as the basis for the diffusion process.
- **Latent Diffusion Model (LDM)**: This module operates in the latent space. It first employs a noising process over a series of time steps. A denoising network is then employed to predict and remove the noise.
- **Conditioning**: This module processes textual description as conditioning data within the model architecture. It is processed by extracting numbers from the sentence in their given order to create a vector of 7 (or 10, in our experiments) statistical features that define the graphs.

The challenge provides a dataset divided into training, validation, and test sets. The training set consists of 8,000 samples, each including a graph and its corresponding textual description, while the validation set contains 1,000 such samples. The test set, on the other hand, is composed of 1,000 textual descriptions without the corresponding graph files. As for the conditioning descriptions, in each are specified 7 properties, namely the number of nodes, of edges, of triangles, of communities, the average node degree, the maximum k core and the global clustering coefficient.

3 Experiments

The code of this project is available at github.com/emilio-pcrd/graph-generation.

With the aim of finding the best possible model for the task, we explored a wide variety of different approaches and architectures. The following section presents a selection of experiments we conducted, in a non-exhaustive way.

First of all, we quickly decided to rule out certain options from the start. For instance, we chose not to explore the use of LLMs or language models to encode the conditioning prompt. The reasons for this decision are that first, the paper [1] indicates that LLMs are not as efficient in this type of scenario and are significantly disadvantaged by their high computational cost. Second, upon analyzing the provided data, we observed that the condition is always presented in a deterministic manner: it consistently comprises the same two structures of sentence, with properties listed in a same fixed order. Consequently, we chose to directly extract the required feature values from the conditions.

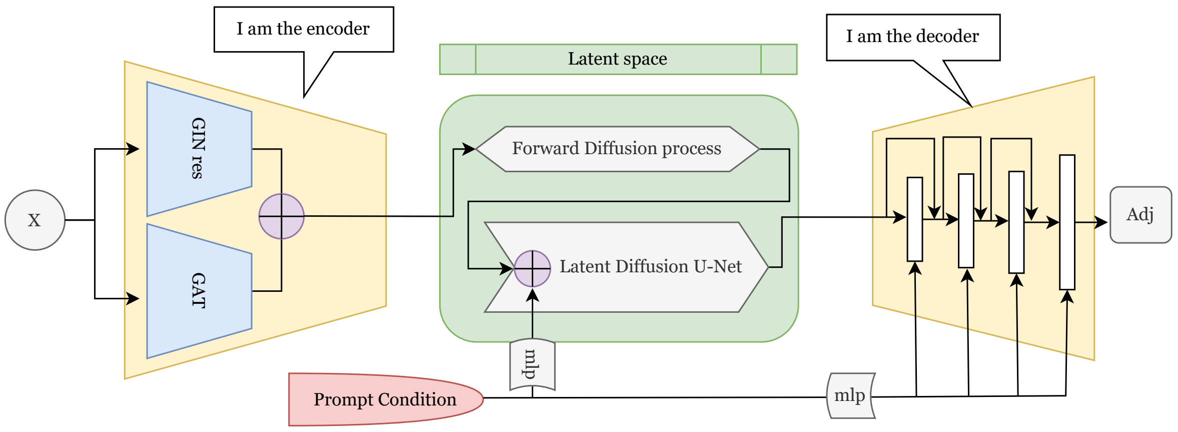

Figure 1: Accrographes Model Architecture

3.1 Feature Augmentation

The conditioning vector originally consisted of only 7 different properties. As a reference, in [1], 15 such properties were available for conditioning. To make our conditions more expressive, and potentially improve model performance, we chose to include the following 3 additional features, as an extraction from the already existing information.

- Density $= \frac{2 \times \text{num\_edges}}{\text{num\_nodes} \times (\text{num\_nodes} - 1)}$. This feature is included in the conditions' features from [1], and can easily be deducted from the available ones in our framework. It helps capture the sparsity or density of the graph.
- Triangle Formation Ratio $= \frac{\text{num\_triangles}}{\binom{\text{num\_nodes}}{3}}$ The idea behind the addition of this feature is that throughout our experiments, we observed that the number of triangles in the generated graphs seemed to be the most unstable property, and was thus susceptible to cause higher error rate during evaluation. Repeating this information, in a way, could potentially help the model's learning process when it comes to the number of triangles.
- Edge-to-Node Ratio $= \frac{\text{num\_edges}}{\text{num\_nodes}}$. We believe that getting the correct number of nodes and edges positively affects the rest of the properties' reconstruction, which is why we repeat the information via this ratio.

These three additional features were calculated during the data pre-processing phase and integrated into the conditioning vector. We consequently modified the hyper-parameter `n_cond = 10`.

3.2 Different Encoder Architecture

In the baseline approach, the encoder used a Graph Isomorphism Network (GIN). To enhance this architecture, we propose a dual-encoder design that integrates both GIN and Graph Attention Network (GAT) encoders. The outputs of these encoders are concatenated and passed through a fully connected layer to reduce the dimensionality from 2×`latent_dim` to `latent_dim`. This approach aims to jointly capture node features, graph structures, and the relative importance of neighboring nodes. By leveraging both encoders, we seek to achieve a more comprehensive representation, which is crucial for the graph generation task.
Inspired by the ResNet architecture ([2]), we also introduced residual connections in the GIN encoder to improve information flow. More specifically, at each output layer of the GIN encoder, the input is added to the output, ensuring better control over the data throughout the encoding process.

During our experiments, we also implemented a PNA encoder (which is available in our code). Unlike GIN or GAT, the PNA encoder uses multiple aggregation functions, making it particularly robust to varying graph degrees by explicitly incorporating node degree information. This enables better generalization on both sparse and dense graphs, such as those that are present in our datasets.
While the approach showed promising theoretical results, it was not realistically computable. Even with a single RTX 4050 GPU (6 GB CUDA memory), the implementation encountered CUDA memory errors, despite efforts to reduce the number of layers, as well as aggregation functions.

3.3 Latent Diffusion U-Net

As seen in previous work ([3], [4]), and specifically inspired by the Stable Diffusion paper ([5]), using a U-Net architecture in the denoiser is particularly well-suited for this task. U-Net uses a symmetric encoder-decoder structure with skip connections, which directly transfer details from the encoder to the decoder. This helps preserve important information during diffusion processes, ensuring high-quality outputs by maintaining better control over denoising. Additionally, it efficiently handles complex data distributions in the latent space. Moreover, the skip connections prevent the vanishing gradient problem, making training more stable.

However, in terms of computational cost, the U-Net denoiser required around eight times more processing time than the proposed model (which is not very environmentally friendly), while offering only marginal performance improvements. Despite this trade-off in efficiency, it achieved our highest Kaggle score at one point, demonstrating its potential. Nevertheless, we opted to retain the base denoiser for subsequent statistical experiments. The only difference is that we opted to change the number of timesteps from 500 to 1000 steps, as proposed in [3] and [5].

3.4 Different Decoder Architecture

A significant improvement to the baseline decoder architecture was achieved by sequentially incorporating the conditioning vector at each layer, transforming it into a conditioning decoder architecture. Additionally, since the last layer of the decoder projects in the adjacency matrix dimension, namely $2 \times \frac{\text{n\_nodes} \times (\text{n\_nodes -1})}{2}$, which is a very high dimension, we smoothed the dimensionality increase across the fully connected layers. We also integrated residual connections, following the same method used in the GIN encoder. As illustrated in Figure 1, this design enhances the information flow during the decoding process.

3.5 Hyper-Parameter Tuning

In our model, we have set the following parameters:

data	baseline generation (8000 for training, 1000 for val, 1000 for test)
epochs autoencoder	600
epochs denoiser	600
n_conditions	7 or 10
optimizer	AdamW
optimizer momentum	$\beta_1, \beta_2 = (0.9, 0.999)$
learning rate	$1e^{-3}$
scheduler	Linear, timesteps=200
batch size	256
denoiser loss	smooth ℓ_1
autoencoder loss	ℓ_1 (reduction = sum)
timesteps diffusion	1000
n_cond denoiser, decoder	128, 64

Table 1: Hyper-parameters tuning for training.

In this study, we experimented with various optimizers and scheduling strategies. Initially, we applied the AdamW optimizer with a linear learning rate scheduler. Later, we tested the `ReduceOnPlateau` scheduler, as described in [3]. However, as we didn't observe any differences during inference, we finally decided to keep the Linear one.
In total, the final model architecture contained over 700,000 learnable parameters.

4 Evaluation and Output Graph Selection

In the context of the Kaggle competition, we did not have access to the test set's graphs. During our experiments, it was therefore difficult for us to assess which methods were the most promising.

To address this issue, we decided to create our own evaluation method. Since the primary objective is to generate graphs with a set of properties that closely match the input conditions, we chose to extract the properties directly from the generated graphs in output and compare them to the ground truth conditions from the test set. Since these properties were on very different scales, it was important to normalize each property, using this basic normalization formula $\frac{\text{prop - mean}}{\text{std}}$. We used this Mean Absolute Error (MAE) as a proxy for the Kaggle score. While this MAE wasn't identical to the Kaggle score (due to the lack of explicit details about how the metric is calculated or how the properties are normalized, and since Kaggle evaluates only 30% of the test set), but it still gave us an idea of how to rank our models' performances.

Over the course of our generations, we started to wonder whether the outputs were consistent, which is a valid concern when being in a generative framework. To try and gain insights on this, we decided to generate multiple outputs for given graphs in order to assess whether our model was producing stable results and whether they aligned with the specified properties.

To investigate this further and compare with the baseline model, we randomly selected a graph condition (graph_8 from the test set) and generated 1000 outputs for this specific graph using both the baseline model and our final model, as detailed in 1. Figure 2 presents the boxplots for the seven different properties.

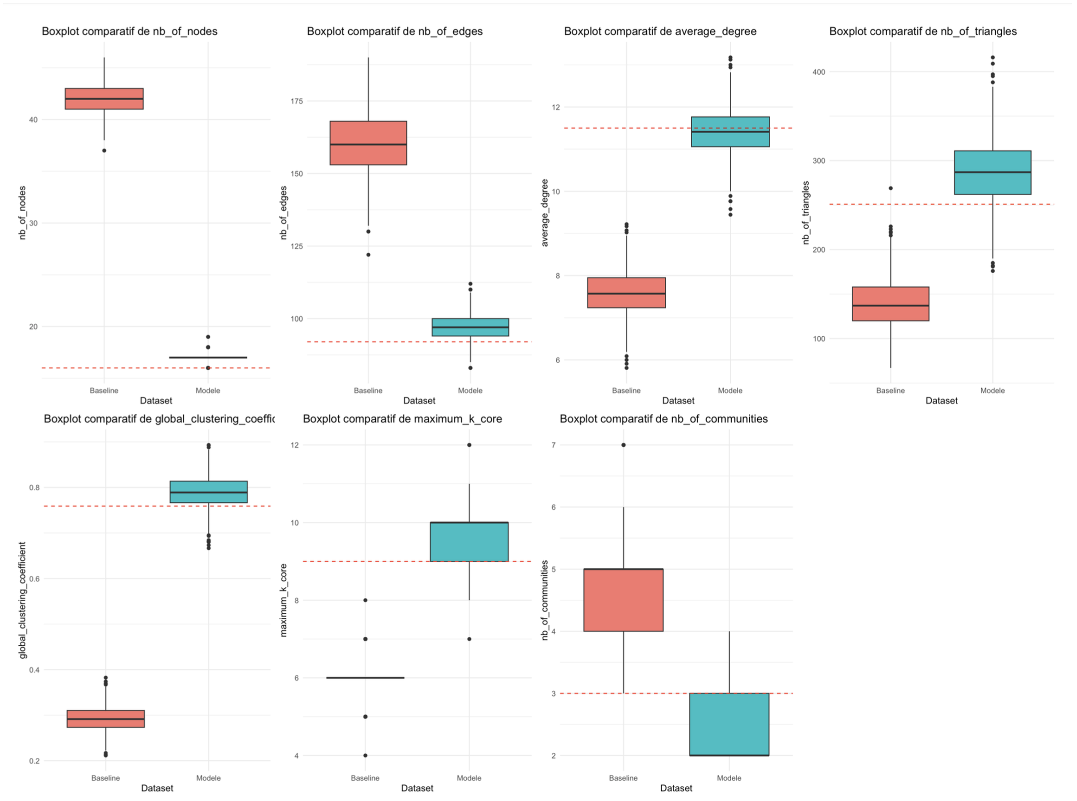

Figure 2: Comparison boxplots for the seven properties of graph_8. In red is the baseline model and in blue our Accrographe's model. The dotted red line represents the ground truth value.

For the provided baseline model, we chose to change the autoencoder's L1 loss function from mean to sum. This adjustment was a key factor in enhancing the baseline model's performance, and we wanted to ensure a fair comparison. Our results show that our model is much closer to the ground truth compared to the baseline model. Moreover, it appears that our model presents some variance in its outputs. Of course, these observations rely on a single graph, but we also conducted this experiments on other random graphs and found comparable conclusions. Moreover, graph number 8 from the test set seems like a representative or 'average' graph.

Using the same generations as for the boxplots, in figure 3, we represent histograms for the three first properties. Once again, we observe that our model's outputs have some variance. This can translate to a form of a instability during inference.

After analyzing our outputs, we concluded that a more effective approach to improve the generations, given an already trained model, is to generate multiple versions of the test graphs, and keep the ones whose properties are the closest to those from the input condition (using MAE again). This strategy, inspired by Bayesian methods like Approximate Bayesian Computation (ABC) [6], aims to reduce the variance in the outputs by selecting the most representative graph according to the target properties. For instance, if we look at the histogram for the number of nodes (on the left of figure 3), we observe that the model gets the correct number wrong most of the time but still occasionally gets it right.

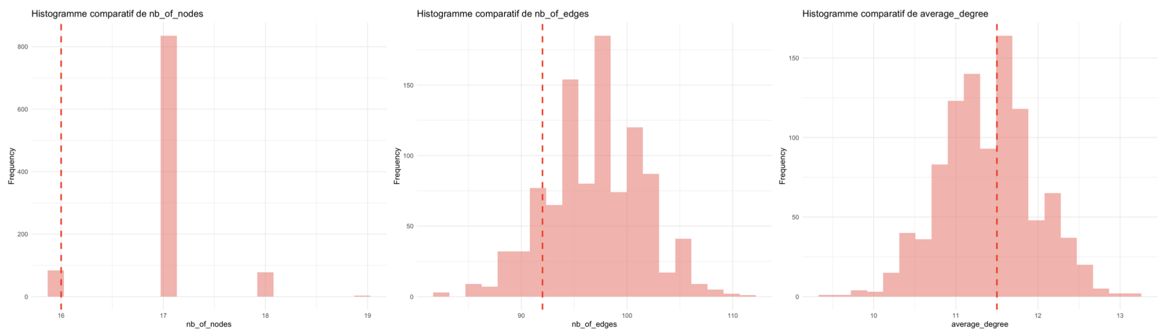

Figure 3: Histograms for graph properties from 1000 generated graph_8: (from left to right) n_nodes, n_edges, average_degree.

In figure 4, we analyze the impact of varying the number of generated graphs on model performance. The plot illustrates the reduction in MAE as the number of generated graphs increases. We finally chose to set the number of generated graphs during post-processing to $n_G = 100$ for computational efficiency.

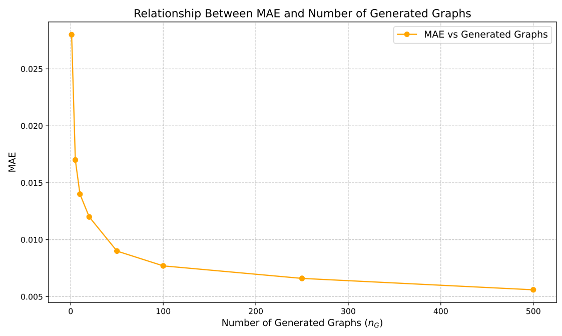

Figure 4: MAE of chosen model for different numbers of generated graphs.

Results

In this section, we present the performance evaluation of our last architectural choices and configurations. Our experiments aimed to identify the most effective setup for minimizing the Mean Absolute Error (MAE) while maintaining computational efficiency. We ended up opting for the architecture featuring the residual

Method	MAE
GIN-GAT + Normal Decoder (n_cond=7)	0.0061
GIN-GAT + Residual Decoder (n_cond=7)	0.0058
GIN-GAT + Residual Decoder (n_cond=10)	**0.0056**

Table 2: MAE results of our last architectures.

decoder. Table 2 presents the results of various architectures with their corresponding MAE loss. The Residual Decoder with our pre-processing (n_cond =10) achieved the best performance for this task.

5 Conclusion

After extensive experimentation, the final model we selected retains the overall structure of the baseline model but introduces key modifications. Specifically, it employs a different encoder architecture with a concatenation of GIN and GAT layers, and a decoder with progressively refined throughout the different layers, and with residual connections. This model finally let us achieve a score of [0.05899] on the Kaggle leaderboard (on approximately 30% of the test set), which we find to be a very promising result.
The main limitation of this model is that it relies on very strong assumptions regarding the input textual conditions. Specifically, if the values of the properties are given in a different order, even if just two values are swapped, it can significantly affect the quality of the generated outputs. However, this issue lies outside the scope of our current problem, and for the task at hand, we have achieved encouraging results.

References

[1] Iakovos Evdaimon, Giannis Nikolentzos, Christos Xypolopoulos, Ahmed Kammoun, Michail Chatzianastasis, Hadi Abdine, and Michalis Vazirgiannis. Neural graph generator: Feature-conditioned graph generation using latent diffusion models, 2024.

[2] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition, 2015.

[3] Javier Nistal, Marco Pasini, Cyran Aouameur, Maarten Grachten, and Stefan Lattner. Diff-a-riff: Musical accompaniment co-creation via latent diffusion models, 2024.

[4] Robin Rombach, Andreas Blattmann, Dominik Lorenz, Patrick Esser, and Björn Ommer. High-resolution image synthesis with latent diffusion models, 2022.

[5] Robin Rombach, Andreas Blattmann, Dominik Lorenz, Patrick Esser, and Björn Ommer. High-resolution image synthesis with latent diffusion models, 2022.

[6] Pierre Pudlo, Jean-Michel Marin, Arnaud Estoup, Jean-Marie Cornuet, Mathieu Gautier, and Christian P. Robert. Reliable abc model choice via random forests, 2015.