MVA

Generating Graphs with Specified Properties

Audrey AIRAUD, Émilie PIC, Garance GÉRARD

2024-2025

1 Introduction

In this project, the objective is to generate graphs that satisfy seven specific properties: the number of nodes, the number of edges, the average degree, the number of triangles, the global clustering coefficient, the maximum k-core, and the number of communities. These properties are provided as input in a text format, where they always appear in a fixed order. The challenge lies in training a machine learning pipeline, leveraging latent diffusion techniques for conditional graph generation, to create graphs that accurately reflect the specified properties while maintaining realistic and meaningful structures.

2 Discovering the Initial Model

The first step of the project was to familiarize ourselves with the provided baseline, based on NGG [2]. This model comprises two main components: a variational graph autoencoder and a latent diffusion model. The autoencoder encodes a graph G into a compressed latent representation z, which is then decoded back into G. The diffusion model operates in this latent space, progressively adding and denoising noise to generate latent representations conditioned on graph properties. The initial implementation of the model, using GIN convolutional layers in the encoder and a denoiser, achieved a Kaggle score of 0.89729.

Through our analysis, we identified that the VAE significantly contributed to the model's errors, showing a lack of precision in its reconstructions. On the other hand, the diffusion model was performing really well in incorporating the conditional vector effectively. Consequently, we decided to focus our efforts on refining the VAE to address its shortcomings and minimize its contribution to the overall error.

3 Redefining the Loss

3.1 Initial Problem

In the initial VAE model, the loss was composed of two components: the Kullback-Leibler (KL) loss and the L1 loss. During training, we observed that the KL loss converged very quickly to 0, while the reconstruction loss stagnated around 0.2. This behavior suggested that the model struggled to optimize both losses simultaneously. The rapid convergence of the KL loss to 0 indicates that its contribution to the overall optimization process was very low, and that it didn't succeeded in structuring the latent space. At the same time, the stagnation of the L1 loss implies that the model failed to improve the reconstruction quality. We thought that these problems in the losses optimization were limiting the model's learning.

3.2 Principal loss adopted

Our goal for the Variational Auto-Encoder was to enable it to capture the principal characteristics of graphs and reconstruct plausible graphs with similar features. To achieve this, we experimented with various loss functions to enhance the model's learning process.

One of the key losses we explored was the binary cross-entropy (BCE) loss. We chose this loss because BCE is commonly used for binary classification tasks, and our adjacency matrices consist of binary values. Essentially, our task can be seen as a binary classification problem, where the goal is to classify each edge in the adjacency matrix as 0 if it does not belong to the graph, and 1 if it does.

This approach significantly improved the model's performance on the test set, achieving a much better score of 0.13074 on Kaggle. While the KL loss exhibited higher values compared to earlier iterations, it eventually stabilized, indicating a balanced trade-off between reconstruction accuracy and latent space regularization.

3.3 Trials on different other losses

Even if the BCE improved greatly our results, it has a problem when trying to improve the model's performances: the model tries to learn the adjacency matrix but lacks sense of global metrics such as the number of edges, nodes, triangles... To try to compensate this issue, we tried a few approaches:

- **Adding components to incorporate the graph global parameters of the graph**: as we experimented that the BCE was trying to capture only the local details of the graph, we tried to add global information of graphs to the loss:

$$\mathcal{L}_{\text{total}} = \alpha \cdot \mathcal{L}_{\text{recon}} + \beta \cdot \mathcal{L}_{\text{degree}} + \gamma \cdot \mathcal{L}_{\text{node}} + \delta \cdot \mathcal{L}_{\text{edge}}$$

We added 3 components to the BCE loss: a component mesuring the difference between the sum of connections for each node in the predicted and true graphs ("degree loss"), a component mesuring the difference of nodes having at least one connection ("node loss"), and a component measuring the difference between the total edge counts in the predicted and true graphs ("edge loss").

The challenge with this loss function was the need to optimize 4 parameters $(\alpha, \beta, \gamma, \delta)$, making a grid search for the best configuration complex. Despite our efforts, we were unable to identify parameter values that resulted in significant improvements. An other problem is that for each loss we perform a mean on the whole batch so it can not work if the batch contains very different graphs.

- **Adding a spectral component** We had the idea of adding a spectral component to the BCE loss: this would compare the predicted graph and the true graph by evaluating their spectral properties through the eigenvalues of their Laplacian matrices. We thought the Laplacian matrix would provide a representation of a graph that reflects its structure and connectivity which add global structural properties to the original loss. However this modification did not help us to improve our score significantly.

- **Adding a contrastive component:** Another idea we had was to help the encoder generate latent vectors that are closer for similar graphs. To achieve this, we attempted to add a contrastive component to the original BCE loss. This component was supposed to play a similar role as the KL, guiding the encoder in structuring the latent space.

  The loss we tested takes as input the embeddings of the graphs and their associated condition vectors to measure their closeness in the latent space (with a cosine similarity). Based on the similarity of the condition vectors, each pair is labeled as either similar or dissimilar using a threshold. For similar pairs, the loss penalizes any differences between their embeddings, encouraging them to be closer in the latent space and conversely for dissimilar pairs. Unfortunately, even with trying to adjust the coefficients in the sum of the BCE component and the contrastive component, we didn't manage to improve our score with this component.

4 Improving the baseline model

4.1 Defining a Metric

Since the metric used in this project is only available on Kaggle, we tried to replicate it. The approach was to compute a Mean Absolute Error, but with adaptations to account for the varying values of the properties. For instance, the number of edges can reach up to 2 500, while the number of nodes is typically limited to 50, and the clustering coefficient is always less than 1. To make the comparison more meaningful and aligned with the Kaggle metric, we normalized these features to create a more logical scale. This normalization helps ensure that each property contributes appropriately to the overall metric. The formula we initially used was as follows:

$$\text{MAE}_{\text{normalized}} = \frac{1}{N} \sum_{i=1}^{N} \sum_{j=1}^{7} \frac{stat\_pred[j] - stat\_true[j]}{stat\_true[j] + \mathbb{1}_{stat\_true[j]=0}}$$

While this formula didn't perfectly align with the metric provided on Kaggle, it served as a reasonable approximation for small values. However, during experimentation, we observed that a slightly different formulation yielded better results. Consequently, we adopted the following adjusted formula:

$$\text{MAE}_{\text{normalized}} = \frac{1}{N} \sum_{i=1}^{N} \sum_{j=1}^{7} \frac{stat\_pred[j] - stat\_true[j]}{stat\_true[j]} \mathbb{1}_{stat\_true[j]\neq 0}$$

4.2 Hyperparameter tuning

To improve our model, we decided to fine-tune its hyperparameters. We initially explored techniques like grid search, varying the main parameters systematically. However, this approach proved to be both time-consuming and computationally expensive, especially given our objective to minimize the metric we designed, which itself requires significant computational effort to evaluate and the high number of parameters to optimize. As a result, we didn't succeed to optimize the default parameters provided by the professor

During our experiments, we observed that adjusting the temperature of the Gumbel-Softmax function had a noticeable impact on the results. Consequently, we focused our efforts on this parameter, testing a range of temperature values at regular intervals between 0.5 and 3 on 20 epochs. Thanks to this approach, we found a best parameter of $\tau = 2$. This helped us to improve our score.

4.3 Forcing the decoder to output the correct number of nodes

In the forward function of the VAE's decoder, we observed that the adjacency matrix was symmetrized before being output. This led us to consider that other transformations could also be applied prior to returning the matrix, in order to force the decoder to produce an adjacency matrix with the desired characteristics. To achieve this, we modified the decoder by incorporating `data.stats` as additional input and constrained the adjacency matrix to reflect a number of nodes consistent with the desired parameters. This modification in the code also helped us to improve our score.

4.4 Selecting the Best Solution from Multiple Generations

Given that we know the metrics we aim to optimize, we can evaluate different graphs to determine which one is the best candidate by computing the loss. Therefore, for each prompt, we generated 100 graphs using the best model available and selected the best one as the final output.

5 Exploration of Alternative Methods

5.1 Testing Different Graph Embeddings on the Baseline Model

The initial autoencoder model uses embeddings derived from the eigenvectors of the Laplacian matrix. To improve the model's efficiency and the quality of the graph representation, we experimented with alternative graph embeddings. For each of these embeddings, we adapted the encoder structure of the initial autoencoder model to accommodate the new representations, aiming to improve the model's performance. Below, we describe the different embeddings explored during this process.

5.1.1 BERT Embeddings

BERT embeddings for graphs are generated by treating the graph's edge list as a text sequence. BERT, being a pre-trained language model, captures relationships between graph elements in a bidirectional manner. This allows it to generate vector representations of the graph in a different embedding space, enabling the model to leverage the contextual information between nodes and edges for improved representation.

5.1.2 DeepWalk Embeddings

DeepWalk generates node embeddings through a random walk approach. The method involves simulating multiple random walks on the graph and then applying the Skipgram model from Word2Vec to the resulting sequences of nodes. This approach facilitates local exploration of the graph's structure, capturing the proximity relationships between nodes. The embeddings generated through this process serve as rich feature representations, which can be used as input to the autoencoder for further analysis.

5.1.3 Testing Prompt Embeddings

In the initial model, the denoiser is provided with features extracted from the prompt describing the graph. To enhance this, we explored generating new embeddings directly from the prompt using BERT. We then compared the performance of the model when fed with the extracted features, the BERT embeddings, and a combination of both.

5.2 GAN-based Methods

Generative Adversarial Networks (GANs) are well-known for their success in image generation. At the beginning of the project, we considered using this type of model to generate graphs. Our approach was to transform our graphs into images; since a graph is defined by its adjacency matrix—a binary matrix of 0s and 1s—it can be represented as a black-and-white image. Using this set of "images," we aimed to train a GAN to generate new images that would correspond to new graphs. Inspired by a GAN model trained on the MNIST dataset (which also consists of black-and-white images), we designed our own GAN model.

However, we encountered a significant challenge: the difficulty of stabilizing the training process. The discriminator consistently outperformed the generator, leading to unbalanced training. While we could have explored this model further, we realized that achieving a well-performing GAN would be far more complex than optimizing the parameters of the baseline model. As a result, we decided not to pursue this approach further.

We also reviewed several papers exploring the use of GANs for graph generation. One approach that particularly caught our attention was described in [1], which utilizes random deep walks for the generator. This method seemed promising and likely more effective than converting graphs into images. However, in the NetGAN paper, the model is

trained on a single graph, whereas our task required working with multiple graphs. This fundamental difference made it challenging to directly apply the NetGAN methodology to our problem.

5.3 LLM

As mentioned in the guidelines, we also attempted to address this problem using a Large Language Model (LLM). However, the results were disappointing. To guide our approach, we drew inspiration from the prompt used in the paper [2] and adapted it to meet our specific requirements. We experimented with one-shot prompting, which was said to yield better performance. An example of the prompt provided to the model is given in B.

When we tested the model on some data, the results were far from the expected format.

To overcome this, we tried generating multiple responses and selecting the best one. One of the main limitations was the time required. While it remained reasonable for small graphs, for larger graphs, the number of edges requested required a significant number of tokens to be generated, which was not compatible with the computational power available.

Additionally, we attempted to fine-tune models like LLaMA or Qwen, but our hardware limitations prevented us from doing so.

Finally, we could have explored creating a new dataset where each entry consists of a prompt, the corresponding edge list of the graph, and the edge list of a graph that does not match the description. Moreover, with this new dataset, we could have employed Direct Preference Optimization (DPO). Indeed, the advantage of DPO is its ability to simultaneously bypass the explicit reward modeling step while avoiding the complexities of reinforcement learning optimization. This approach could have helped avoid the memory problems we faced during fine-tuning.

5.4 Resolution via Combinatorial Optimization

Two of the three members of our group are pursuing a Master's degree in Operations Research. This motivated us to compare our results with those obtained using a combinatorial optimization method, Integer Linear Programming. This method consists of minimizing an objective function with integer variables under linear constraints.

The problem we studied can be reformulated as minimizing the absolute difference between the true and predicted features, under the constraint that the output must be a valid graph. Given that some features are more complex to model than others, we focused on the simpler ones: the number of nodes, the number of edges, the number of triangles, and the average node degree. The detail of the model is given in C.

This model produces satisfactory results for small graphs (≤ 25 nodes). However, it takes significantly longer to find good solutions for larger graphs. To manage computation time, we impose a 20-second limit per graph.

6 Summary of Results

We have summarized the various results in terms of losses and scores in the table below. As shown, the scores achieved with the BCE loss are significantly better than those with the other losses. However, the choice of encoder does not seem to have a major impact on the results. The most significant factor influencing the score appears to be the loss function used.

Method	**Train RL**	**Val RL**	**MAE**	**Score Kaggle**
Initial Model	0.2188	0.2150	1.4421	0.89729
BERT embedding	0.2189	0.2214	1.4927	0.88806
GINConv + BCE	0.0818	0.0887	0.1365	0.13074
DW + BCE	0.0825	0.0888	0.1342	0.13823
$\tau = 2$ + BCE	0.0880	0.0935	0.1225	/
$\tau = 2$ + BCE + best in 100 generated	/	/	0.0334	0.05907
$\tau = 2$ + BCE + loss nb nodes	0.0837	0.0909	0.1073	0.11124
$\tau = 2$ + BCE + loss nb nodes + best in 100 generated	/	/	0.0282	0.05179
Linear Programming	N/A	N/A	N/A	0.52096

Table 1: Comparison of loss and performance on Kaggle with different methods

Some curves have been included in the appendix A to visualize the performances.

7 Conclusion

This project has been an enriching experience, allowing us to experiment with generative models and gain a deeper understanding of diffusion models and variational autoencoders through experimentation.

Working with graphs as the central focus of this project was a challenge in itself, as it required the model to capture global structural information, which is a challenging task.

Additionally, the project taught us the importance of thinking critically before acting, as each test or modification to the model required significant time to complete. Over the course of the project, we learned to carefully plan and evaluate our decisions before initiating operations.

One goal we would have liked to achieve, but didn't have time for, was to develop an efficient way to optimize hyperparameters, both for the autoencoder and the denoiser. Similarly, we would have liked to design a more effective loss function that could minimize the MAE more efficiently. Lastly, we also explored other avenues for improvement, such as modifying the structure of the encoder and decoder, which seems a promising direction for future work.

A Results Visualization

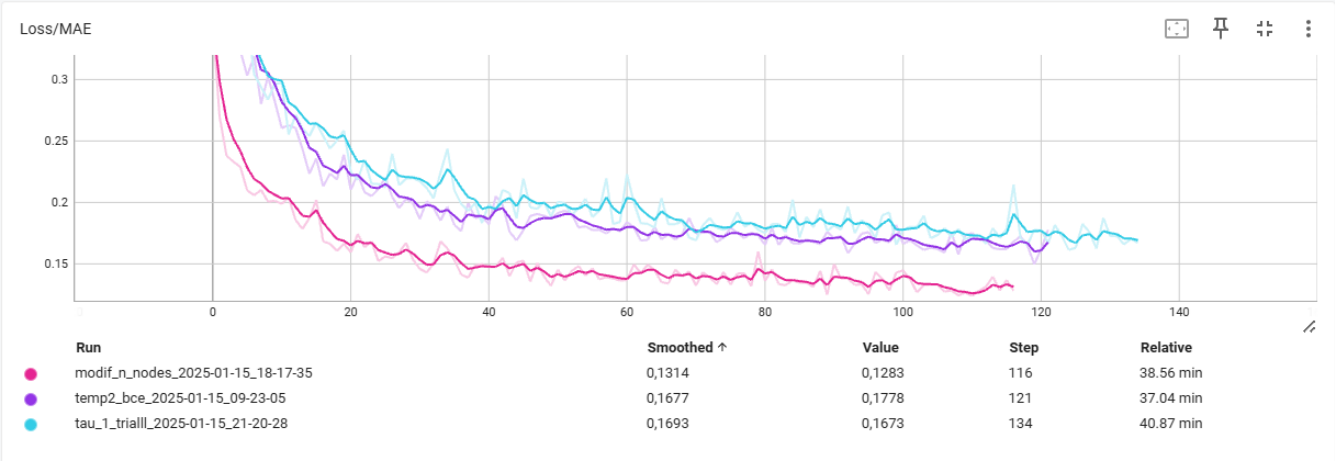

Figure 1: Performance of the autoencoder with the MAE loss across three different experiments. The blue curve represents $\tau = 1$ for the Gumbel-Softmax, the violet curve represents $\tau = 2$ for the Gumbel-Softmax, and the pink curve corresponds to $\tau = 2$ with the number of nodes explicitly enforced by the decoder.

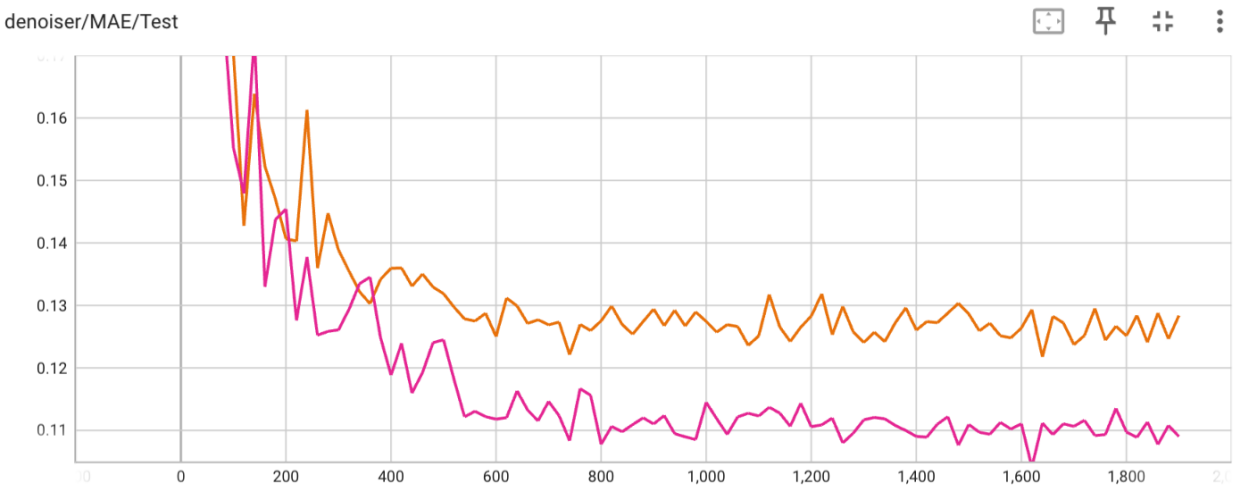

Figure 2: Performance of the MAE after the denoiser. The pink curve represents $\tau = 2$ with the number of nodes explicitly enforced by the decoder, while the orange curve corresponds to $\tau = 2$ without enforcing the number of nodes.

B Prompt example for LLM

Prompt Example

```
<SYSTEM>We represent an edge list like this (1, 0), (0, 3), (1, 2), (2, 3) where a pair (1,
0) indicates an edge between node 1 and node 0 without weight.  The pair (i, j) and the pair
(j, i) are the same so use only one of them and the edge (i, i) is not allowed.  when you
answer the question answer with the edge list only and you should provide the same number of
edges and respect the number of nodes and don't give explanation and don't print this , ...  ,
print all the edges and respect the edge format.  </SYSTEM>

Example:  <USER>Create an edge list for an unweighted graph with 10 nodes, 5 edges, 1.0
average degree, 0 number of triangles, 0.0 global clustering coefficient, 1 maximum k-core
and 4 communities?</USER> <ASSISTANT> Here is the edge list:  (0, 1), (5, 6), (3, 9), (7, 8),
(2, 4)</ASSISTANT> Question:  <USER>Create an edge list for an unweighted graph with 12 nodes,
37 edges, 6.17 average degree, 38 number of triangles, 0.55 global clustering coefficient, 5
maximum k-core and 2 communities?</USER> <ASSISTANT> Here is the edge list:
```

C PLNE model

Variables We introduce the following variables:

- $x_{i,j}$: A binary variable representing the existence of an edge between nodes i and j.
- n_i: A binary variable representing whether a node i exists.
- $t_{i,j,k}$: A binary variable representing whether a triangle exists between nodes i, j, and k.

Additionally, we define integer variables representing the absolute differences between the predicted and actual statistics for number of nodes, edges, triangles, and average degree.

Objective Function We want to minimize the difference between the true and predicted feature. Thus, the objective function is (and we want to minimize it):

$$\text{dif}_{\text{nodes}} + \text{dif}_{\text{edges}} + \text{dif}_{\text{triangles}} + \text{dif}_{\text{avg\_degree}}.$$

Constraints This model includes two types of constraints: those that ensure the variables accurately represent a valid graph with the correct number of triangles, and those that guarantee the proper calculation of differences in the model's statistics.

Symmetry of Edges: Since the graph is undirected, an edge between i and j implies an edge between j and i:

$$\forall i, j \in [\![1, n]\!],\ i \neq j, \quad x_{i,j} = x_{j,i}.$$

Node Existence A node exists if it has at least one edge connected to it:

$$\forall i \in [\![1, n]\!], \quad n_i \geq \frac{\sum_{j=1}^{n} x_{i,j}}{50},$$

$$n_i \leq \sum_{j=1}^{n} x_{i,j}.$$

Triangle Existence A triangle exists between nodes i, j, and k only if all three corresponding edges exist. This is enforced by:

$$\forall i, j, k \in [\![1, n]\!],\ i < j < k, \quad \begin{cases} t_{i,j,k} \leq x_{i,j}, \\ t_{i,j,k} \leq x_{j,k}, \\ t_{i,j,k} \leq x_{k,i}, \\ t_{i,j,k} \geq x_{i,j} + x_{j,k} + x_{k,i} - 2. \end{cases}$$

Statistical Differences The following constraints ensure that the variables representing absolute differences accurately correspond to the actual differences. These constraints can be reformulated as linear expressions, making them compatible with the solver for efficient computation.

$$\begin{cases} \left| \sum_{i=1}^{n} n_i - n_{\text{nodes}} \right| \leq dif_{\text{nodes}}, \\ \left| \sum_{i=1}^{n} \sum_{j=i+1}^{n} x_{i,j} - n_{\text{edges}} \right| \leq dif_{\text{edges}}, \\ \left| \sum_{i=1}^{n} \sum_{j=i+1}^{n} \sum_{k=j+1}^{n} t_{i,j,k} - n_{\text{triangles}} \right| \leq dif_{\text{triangles}}, \\ \left| \frac{\sum_{i=1}^{n} \sum_{j=i+1}^{n} x_{i,j}}{n} - avg_{\text{degree}} \right| \leq dif_{\text{avg\_degree}} \end{cases} \tag{1}$$

References

[1] Daniel Zügner Aleksandar Bojchevski, Oleksandr Shchur and Stephan Günnemann. Netgan: Generating graphs via random walks. *ICML*, 3, 2018.

[2] 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.