Project Report: Generating Graphs with Specified Properties

Baptiste Dupré, Leo Stepien, Yassine Machta

January 15, 2025

Abstract

This report outlines the approach and methods used in the project : "Generating Graphs with Specified Properties" in the context of our Master course "Advanced AI for text and graphs". We will present how we studied the data, our initial exploration, and the main solutions we chose. We will present a few other methods that did not yield results.

1 Introduction

In this project, our aim is to learn to generate graphs using their properties as input. In this project, we will be interested in 7 graph properties.

- Number of nodes
- Number of edges
- Number of triangles
- Average degree of the graph nodes
- Graph max-k-core
- Global clustering coefficient
- Number of communities

For this project, we have a training dataset of 8000 graphs, a validation dataset of 1000 graphs, and a testing dataset of 1000 graphs.

As we can see in Fig.1, we have plotted the distribution for these graph parameters across our datasets and will not suffer from any out-of-distribution problem. This means that we will not have to generate or add other graphs to fill in any blanks. We also not that the maximum number of nodes to be generated are 50. By curiosity, we also generated 100 000 graphs and they had roughly the same feature distribution

2 Data preparation and given codebase exploration

The training and validation dataset have the following 3 tensors that will be the base of our input into our model:

- Node-level features: By default, obtained from a spectral decomposition algorithm
- Graph-level features: a tensor of 7 values describing the main properties mentioned above

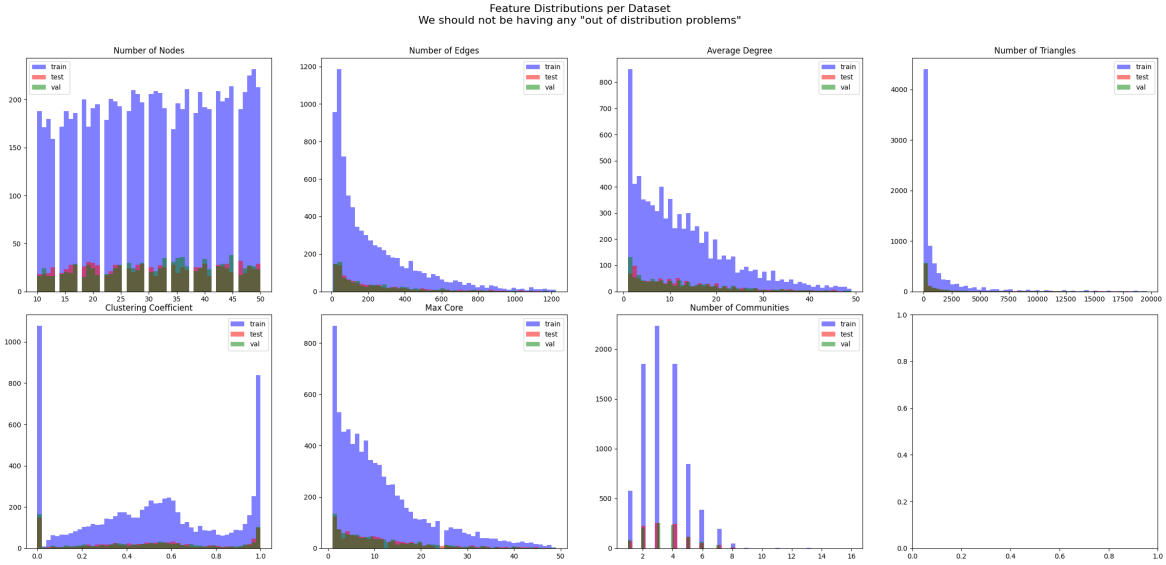

Figure 1: Distribution of feature values across training, testing and validation datasets

- Graph adjacency matrix: The raw adjacency matrix, unnormalized and with no self-cycles for the nodes (1s in the diagonal)

Naturally, the test set only contains the graph-level features (ie. the prompted properties).

We have also studied the initial Variational autoencoder and the latent diffusion model given to us.

The base model $m(G_G \mid P_G)$ that generates a "Generated graph" G_G given the "Properties of the graph" P_G and node features "Properties of the nodes" P_N. A generative model assumes there exists latent Z such that

The initial proposed model consisted of three ingredients:

1. An encoder, $q_\phi(Z)$, which maps graphs to a latent variable Z. It does the mapping by using the training graph's adjacency matrix and node-level features in the encoding process.

2. A decoder, $p_{\theta'}(G_G \mid Z)$, which maps the latent variable to a graph.

3. The latent variable Z is sampled using the reparameterization trick: $Z = \mu + \sigma \cdot \epsilon$, where μ is the mean of the distribution $p(Z \mid P_G)$ learned by the encoder, $\sigma = \exp\left(\frac{\log(\sigma^2)}{2}\right)$ is the standard deviation learned from the log-variance, and $\epsilon \sim \mathcal{N}(0, I)$ is sampled from a standard normal distribution. During evaluation/testing the model uses the mean μ directly.

The loss function in this Variational Autoencoder (VAE) consists of two main components: the reconstruction loss and the Kullback-Leibler divergence (KLD) regularization.

1. **Reconstruction Loss:** This is computed as the mean L1 loss between the reconstructed adjacency matrix $\hat{A}$ (produced by the decoder) and the original adjacency matrix A, which represents the graph structure. It is given by:

$$\mathcal{L}_{\text{recon}} = \frac{1}{N} \sum_{i=1}^{N} \left| \hat{A}_i - A_i \right|$$

where N is the number of entries in the adjacency matrix.

2. **Kullback-Leibler (KL) Divergence:** This term regularizes the latent variable distribution to be close to a standard normal distribution. It is computed as:

$$\mathcal{L}_{\text{KL}} = -\frac{1}{2}\sum_{i=1}^{N}\left(1 + \log(\sigma_i^2) - \mu_i^2 - \sigma_i^2\right)$$

where μ_i and σ_i^2 are the mean and variance of the latent variables computed by the encoder for each graph.

The final loss function is a weighted sum of the reconstruction loss and the KLD:

$$\mathcal{L} = \mathcal{L}_{\text{recon}} + \beta \cdot \mathcal{L}_{\text{KL}}$$

where β is a hyperparameter that controls the trade-off between reconstruction quality and regularization.

3 Methods

3.1 Concatenating Graph-level features

We quickly noticed the absence of the Graph level features in the models given to us (which had a base MSE score of 0.80). Our first instinct was to concatenate these graph level features to the decoder and to the encoder in order to obtain new estimators:

1. An encoder, $q_\phi(Z) \mid P_G$, which maps graphs AND the graph-level features to a latent variable Z.

2. A decoder, $p_{\theta'}(G_G \mid Z, P_G)$, which maps the latent variable and the concatenated graph-features to a graph.

This caused a jump in our results from 0.80 to 0.20 with no real difference in training time.

3.2 Modifying loss functions to punish reconstructed graphs' features

The ambition for this idea was to simply compute the generated graph's features and to supervise by them in order to make sure the decoder generated graphs as faithful as possible to the input parameter prompts. The hiccup is to find differentiable methods to compute these parameters as we still need to use torch tensors without braking the gradient chain.

1. Number of edges: as the model predicts a 50x50 adjacency matrix containing the edges. It's very quickly possible to determine the number of edges by summing the upper triangular part of the adjacency matrix

2. Avg degree: This follows the same concept since we have to sum the edges by line (or row) to obtain the degree of each node that we can then average.

3. Number of nodes: The most straightforward way to do this is to teach the model to predict self-cycles. Meaning for a graph of 21 nodes, The first 21 elements of the diagonal of the adjacency matrix will cointain 1s. This also changes our initial data processing. This solution introduces the following problems.

4. Number of triangles : We noticed in our MSE that the error for n_triangles was always the largest. Maybe adding a term that penalizes it will improve training we compute the number of triangles by summing the trace for adj^3 and dividing by 6.

5. Edge-node coherence: Since we learn to predict self cycels we expect that $m_{i,j}$ of the adjacency matrix is equal to 1 then that means that $m_{i,i}$ and $m_{j,j}$ must also contain 1. our experiments showed that wasn't also the case.

This was our initial iteration of these losses. We quickly realized that weighing the loss with the original losses was also problematic as some were so much bigger than others.

$$\mathcal{L}_{\text{penalization}} = MSE(n\_nodes)+MSE(n\_edges)+10^{-3}*MSE(n\_triangles)10^{-4}*+Edge\_node\_coherence()$$

$$\mathcal{L} = \mathcal{L}_{\text{recon}} + \beta \cdot \mathcal{L}_{\text{KL}} + \beta_{penalization}\mathcal{L}_{\text{penalization}}$$

This caused the model to have convergence issues and we couldn't get a result out of this one.

3.3 Attention layers

Taking inspiration from our lab work we also tried to explore the impact of attention networks in the Encoder. The easiest way to do this is to use

The GATv2 operator from the paper *"How Attentive are Graph Attention Networks?"* improves upon the standard Graph Attention Network (GAT) by addressing the static attention problem.

In GATv2, the attention mechanism is defined as follows:

$$\mathbf{x}'_i = \alpha_{i,i}\Theta_s\mathbf{x}_i + \sum_{j\in\mathcal{N}(i)} \alpha_{i,j}\Theta_t\mathbf{x}_j$$

where $\alpha_{i,j}$ are the attention coefficients, computed as:

$$\alpha_{i,j} = \frac{\exp\left(\mathbf{a}^\top \text{LeakyReLU}(\Theta_s\mathbf{x}_i + \Theta_t\mathbf{x}_j)\right)}{\sum_{k\in\mathcal{N}(i)\cup\{i\}} \exp\left(\mathbf{a}^\top \text{LeakyReLU}(\Theta_s\mathbf{x}_i + \Theta_t\mathbf{x}_k)\right)}$$

GATv2 allows us to use graph convolutions along side edge features. We can create edges features by concatenating the node features for the two nodes of the edge.

3.4 Augmenting node-level features

Besides the features stemming from Spectral embedding we also manually added these features. This improved our score by .05 points from 0.2 to 0.15

Features of Interest:

1. Degree of node
2. Sum of degrees of the neighborhood of a node
3. Number of nodes in the connected component
4. Number of edges in the connected component
5. Number of triangles where the node is involved
6. Self-loop existence (is there a path that comes back to this node ?)
7. Local clustering coefficient
8. Core number

This improves our performance if we don't use attention from 0.20 to 0.15ish aswell. But when coupled with attention we get a marginal improvement to 0.13

3.5 Contrastive Learning

To ensure that graphs with similar properties have similar representations in the latent space, we implemented a contrastive learning approach. Upon loading the graphs we use K-Means algorithm to assign a cluster to the graphs based on their features. This method enforces closeness in embeddings for graphs within the same cluster while maximizing separation between

embeddings of graphs from different clusters.
A contrastive loss is computed using cosine similarity on the latent embeddings produced by the encoder and ensures separation of these embedding between clusters by minimizing the cosine distance between embeddings of the same cluster and enforce a margin when clusters differ.

3.6 Integrating Gaussian Mixture Variational Autoencoders (GMVAEs) for Latent Space Alignment

Building upon the idea of leveraging contrastive learning to separate embeddings in the latent space, which yielded good results, we explored the Gaussian Mixture Variational Autoencoder (GMVAE) framework. By modeling the latent space as a Gaussian Mixture, we expected each cluster in the latent space to correspond to a distinct Gaussian distribution, facilitating clearer separation and better alignment of clusters.

This approach is inspired by the work in [4], where a GMVAE is used for classification purposes. The method presented in [4] combines a GMVAE that maps class labels to different latent Gaussian distributions and a VAE that reconstruct class labels from objects. They then take the output labels of the VAE and embed them with the GMVAE and use a contrastive loss to force the representation of labels generated by the VAE to be distributed according to the Gaussian Mixture of the GMVAE latent space. We kept this idea but working with feature clusters, we pretrained a model to go from an adjacency matrix to the cluster label and used it to determine the label of the graph generated by our original VAE. Unfortunately, we did not observe a major improvement in our previous methods.

3.7 Constraining denoising Learning

Our only attempt to improve the denoiser was to also try to supervise the denoiser by the graph reconstruction loss. starting from the graph's encoding in the latent space, we add noise to create a gaussian distribution that we can sample to and then denoise back to the original latent space using the Denoiser. The loss for this model is how accurate is the predicted noise compared to the original one. We also thought we'd used the predicted noise to get back th=o the predicted latent representation, decode it and also supervise with the graph recon loss. Since we use the autoencoder's decoder we tried this in eval mode and in train mode and in both cases there was no serious impact on our performance.

4 Conclusion

In this project, we integrated various established methods and techniques commonly employed in the development of graph neural networks, leveraging the existing architecture and improving its performance. By doing so, we not only improved the model's ability to address the objective but also gained a deeper theoretical and practical understanding of the problem and its solution. We approached this project in an exploratory spirit, implementing ideas one after another to refine our understanding and improve the model. Along the way, we uncovered several promising directions for future work. For instance, adding graph-convolutional layers to the decoder could enhance its expressiveness and performance [3]. Similarly, re-imagining the architecture, such as introducing a denoiser that directly generates adjacency matrices, presents an exciting opportunity to push the model further.
We are pleased with the progress in our understanding and remain inspired by the potential of other ideas to advance the model's capabilities even further. We also had fun figuring out how to deploy the model on HuggingFace Spaces. Github

References

References

[1] Evdaimon, I., Nikolentzos, G., Xypolopoulos, C., Kammoun, A., Chatzianastasis, M., Abdine, H., & Vazirgiannis, M. (2023). Neural Graph Generator: Feature-Conditioned Graph Generation using Latent Diffusion Models. *arXiv preprint arXiv:2403.01535*. https://doi.org/10.48550/arXiv.2403.01535

[2] Yuanqidu. (2025). Awesome-Graph-Generation: A curated list of up-to-date graph generation papers and resources. GitHub repository. Retrieved from https://github.com/yuanqidu/awesome-graph-generation

[3] Chattopadhyay, A., Zhang, X., Wipf, D. P., & Vidal, R. (2022). Structured Graph Variational Autoencoders for Indoor Furniture Layout Generation. *arXiv preprint arXiv:2204.04867*. https://doi.org/10.48550/arXiv.2204.04867

[4] Bai, J., et al. (2022). Gaussian Mixture Variational Autoencoder with Contrastive Learning for Multi-Label Classification. *arXiv preprint arXiv:2112.00976*. https://doi.org/10.48550/arXiv.2112.00976