Lehrstuhl für Data Science

Deep Auto-Regressive Machines for Graph Construction

Masterarbeit von

Abhishek Subedi

1. Prüfer	2. Prüfer
Prof. Dr. Michael Granitzer	Prof.-Dr. Harald Kosh

April 18, 2023

Contents

Abstract

Deep generative models for graph generation are crucial for exploring physical phenomena, social sciences, and biological networks. This research investigates deep auto-regressive models for the graph generation. The research introduces novel techniques such as graph deconstruction/construction, graph deconstruction tree model, unparametrized construction sequences, complex decision operation or states, extended deep state-machine with multiple states, and learning complex graph embedding in a single state. The results show that an extended deep-state machine is a possibility that allows graph construction with fewer and more complex states. However, extended deep-state machines were not able to generate statistically similar graphs to the base model. Further improvements in construction sequence that resembles the distribution of the target graphs, or move in the direction of reinforcement learning are some possible future works.

Acknowledgments

I would like to express my deepest appreciation to all who have supported me throughout the completion of my Master's Thesis.

My most profound appreciation goes to my supervisor, Julian Stier, for giving me the opportunity to be a part of this exciting research. I sincerely thank him for his guidance, which has significantly enhanced my research skills. The research insights I received have motivated me to further continue research and development activities in the future.

I am privileged to do research in the Chair of Data Science and am extremely thankful to Prof. Dr. Michael Granitzer for providing me with this opportunity. I am grateful to Prof. Dr. Harald Kosh, for agreeing to be part of my thesis committee.

Finally, I would like to thank my dear family, who made my journey to the University Of Passau a reality. Their unwavering love, support, and encouragement gave me the strength I needed. A special thanks to my amazing friends whose time, laughter, and moments of joy added fun to my journey.

List of Figures

List of Tables

1 Introduction

1.1 Motivation

A graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$ is a combination of nodes $v \in \mathcal{V}$ and edges $e \in \mathcal{E}$. It has the necessary properties that can represent large complex systems such as social networks, protein interaction networks, knowledge graphs, natural systems, and various other problems that have highly interrelated information between the elements of the problem. The expressive nature of graphs e.g. a protein-protein network can be represented in a learned model, that can be utilized in the prediction of new proteins or protein interfaces [Zho+20]. Graph neural networks (GNNs) are models to capture the relationships and information of nodes, edges, and graphs as a whole. Evolved from an underlying principle of deep learning, graph neural networks have shown phenomenal performance on various tasks such as graph classification, node prediction, link prediction, etc. [Zho+20].

A simple architecture of graph neural networks consists of two basic steps: Message passing with the refinement of vector representations of vertices and edges, and aggregation of these representations. As shown in figure (2.2), each node v creates a message $\mathbf{m}_v$ to be sent to other nodes, and each node v aggregates the messages from its neighboring nodes. To add expressiveness, non-linearity activation functions such as ReLU, and Sigmoid can be added in message and aggregation steps. The output of the architecture is an embedding space that represents the input[YYL20].

Moving a step ahead of tasks such as classification and prediction, graph generation is the next important piece of work that is fundamental for new research and explorations in engineering, biology, and social sciences. Generating synthetic graphs similar to real graphs can help in the discovery of a new drug, design material, model social networks, construct knowledge graphs, etc. [You+18]. Recent developments in deep generative models of graphs have given the ability to learn the graph formation process. These models can learn from the data and apply the learned model in discovering new graph structures and completing the expanding graphs

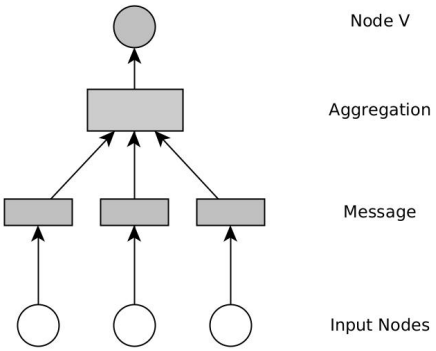

Figure 1.1: A Simple Architecture of GNN

[You+18]. The existing methods provide various techniques to solve the problem [YYL20] [Li+18], but learning a model from a set of graphs and utilizing it to generate realistic graphs is a challenge. This research investigates a new approach to the problem of graph generation.

1.2 Research Gap

Previous studies show, generating graphs in an auto-regressive model is a very efficient approach [SG21] [You+18]. There are still open questions that need to be experimented with and find more ways to expand the boundaries of methods for graph generative models. Here we focus on the limitations of the previous two research, Learning Deep Generative Models of Graphs (DGMG) [Li+18] and Deep Graph Generator (DeepGG) [SG21]. DGMG [Li+18] and DeepGG [SG21], both model incorporates simple decision operations, adding a node f_{add_node} and adding an edge f_{add_edge} to construct a graph. These decision operations are limited to the generation of a single node and a single edge in each step of graph generation.

This research investigates how complexity can be added to decision operations and how well a deep auto-regressive model learns these complexities and generate graphs of similar structures. Instead of having two simple decision operations, more than two complex decision operations are designed that are capable of adding a complex structure of nodes and edges in a single step to the existing graph. This will overcome the shortcoming of previous approaches and will allow the model to learn small complex structures within the graph and generate graphs in fewer steps. The series of these decision operations add up to form a sequence that represents a graph, termed as ***graph construction sequence***.

DGMG [Li+18] obtains its construction sequence through breadth-first-search (bfs) traversal and depth-first-search (dfs) traversal whereas, DeepGG [SG21] considers (bfs), (dfs) and the original construction process of any scale-free graph. Hence both above-mentioned models are biased toward scale-free networks. To overcome the bias, a different method called the ***graph deconstruction tree model*** (3.3.3) is envisioned for sequence generation.

To investigate a different method than DeepGG, DGMG, or GraphRNN, a new methodology where inverse operations of the decision operations are designed. These inverse operations are inverse to the existing decision operations and are used to deconstruct a graph until they are an empty graph. In the process of deconstruction, a series of inverse operations are applied that create a sequence termed as ***graph deconstruction sequence***. Each inverse operation in the deconstruction sequence is replaced with its corresponding decision operation and the whole sequence is reversed. This finally gives us a construction sequence that consists of decision operations to recreate the same graph. The whole process is done with a graph deconstruction tree model envisioned for sequence generation. The sequence created in this phenomenon is ***unparameterized sequence***, **e.g. [op1, op2, op3, op4,...]**. As it contains only the decision operations and no information regarding which nodes are to be connected. Parameterized sequences are trained with auto-regressive models to generate new samples of graphs.

1.3 Research Questions

This research focuses on learning local graph operations and experiments with deep auto-regressive models. These generative models of graphs are to be more flexible and robust in graph generation. Following are research questions that are experimented with and explored throughout the research.

- **Can we extend/generalize the idea of DGMG of a state machine into more states?**
- **How well does the auto-regressive model of graphs learn the extended state machine model?**
- **Compare the new graph generative model with existing graph generators such as DeepGG and DGMG.**

2 Technical Background

This section is concerned with the technical and algorithmic methods for graphs, neural networks, and graph neural networks. The properties of graphs, such as degree, centrality, and degree distribution or power-law degree distribution are emphasized as it provides vital information to understand the characteristics, and types of graphs (2.1). Section (2.2) focus on the basic architecture of neural networks, its optimization procedure, and the backpropagation for improving the learning algorithm. This concept (2.2) has given a base to design a neural network that is specific to graphs known as a graph neural network (GNN). Section (2.3) describes the importance, application, and various tasks of graph neural networks. It gives insight into the working mechanism of GNN such as neural message passing, graph pooling, generalized message passing, message passing with self-loops, and neighborhood normalization including the basic architecture of GNN and Graph Convolution Networks (GCNs).

2.1 Graphs

Graphs are data structures and common languages for modeling complex systems. Graphs represent objects and their relations in terms of nodes and edges respectively. Mathematically, a graph $\mathcal{G}(\mathcal{V}, \mathcal{E})$ is a set of vertices $\mathcal{V}$ and a set of edges $\mathcal{E}$ that exists between the vertices. An edge connecting two vertices $u \in \mathcal{V}$ and $v \in \mathcal{V}$ is denoted as $(u, v) \in \mathcal{E}$ [Ham20]. The number of vertices of a graph $\mathcal{G}$ is referred to as the *order*, denoted by $|\mathcal{G}|$, and the number of edges written as $||\mathcal{G}||$.

Among various graph prop Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ and $\mathcal{G}^{'} = (\mathcal{V}^{'}, \mathcal{E}^{'})$ are two graphs. Graph $\mathcal{G}$ and $\mathcal{G}^{'}$ are called isomorphic graphs and written as $\mathcal{G} \simeq \mathcal{G}^{'}$, when there is a bijection $\varphi : \mathcal{V} \to \mathcal{V}^{'}$ with $xy \in \mathcal{E} \Leftrightarrow \varphi(x)\varphi(y) \in \mathcal{E}^{'}$ for all $x, y \in \mathcal{V}$. This kind of mapping φ represents isomorphism. Different classes of graphs such as triangles can be closed under isomorphism which is called graph property. For example, if a graph $\mathcal{G}$ has three pairwise vertices adjacent to each other, then all the graph isomorphic to $\mathcal{G}$

will have the same nature. Hence, a mapping that takes graphs as arguments can be termed graph invariant, if it resembles equal values to the existing isomorphic graphs $\mathcal{G}$ [Die06].

2.1.1 Degree

Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be a graph with several nodes and edges. The set of neighbouring vertex v in $\mathcal{G}$ is given as $\mathcal{N}_{\mathcal{G}}(v)$. An important property of a vertex relies on its degree, it denotes the number of links connected to other vertices from vertex i. The degree $d_{\mathcal{G}}(v) = d(v)$ of a vertex is equal to the number of edges $|\mathcal{E}(v)|$ at v. Formally, k_i is denoted as the degree of the i^{th} vertex in the graph [Bar13]. A vertex having degree 0 is not connected to any other vertex and is isolated. The number $\delta(\mathcal{G}) := \min \{d(v)|v \in \mathcal{V}\}$ is the minimum degree for the graph $\mathcal{G}$ and similarly $\Delta(\mathcal{G}) := \max \{d(v)|v \in \mathcal{V}\}$ is the maximum degree for the graph $\mathcal{G}$. Degree information is an important property of a graph as it can give an insight into the structure of graphs.

Degree Distribution

A graph can have an extremely complex structure since the edges and nodes create complicated patterns. The complex formation of graphs using simple measures can capture the characteristics of graphs in a clear way. One such measure is degree distribution p_k, which gives the probability of a vertex chosen randomly from a graph having degree k. The probability p_k, in the normalized form is given as [Bar13],

$$\sum_{k=1}^{n} p_k = 1 \tag{2.1}$$

It gives a frequency distribution of the degree sequence. Here p_k is the fraction of vertices in a graph having degree k. In other words, when a vertex is randomly selected, it will have degree k where probability is p_k. A highly connected vertex with a high degree is called a hub. Degree distribution does not capture the whole structure of the graph but can give a raw estimation of vertices of degree k.

Scale-Free Graphs and Power Law

A network composed of large hubs is characterized as a Scale-Free network. These networks follow power-law degree distribution. For an undirected graph, power-law distribution is given as

$$\mathcal{P}_k \sim k^{-\gamma} \tag{2.2}$$

Where γ is the degree exponent and $\mathcal{P}_k$ decays gradually with the increase of degree k [Bar13]. The degrees of the nodes in a graph are positive, $k = 0, 1, ..., n$. The discrete rule defines that a node has k links with probability $\mathcal{P}_k$ and given as

$$\mathcal{P}_k = \mathcal{C}k^{-\gamma} \tag{2.3}$$

Here $\mathcal{C}$ is a constant, obtained by normalization and we know that,

$$\sum_{k=1}^{\infty} \mathcal{P}_k = 1 \tag{2.4}$$

From (Equation 2.3, 2.4) we obtain,

$$\mathcal{C} \sum_{k=1}^{\infty} k^{-\gamma} = 1 \tag{2.5}$$

$$\mathcal{C} = \frac{1}{\mathcal{C} \sum_{k=1}^{\infty} k^{-\gamma}} = \frac{1}{\zeta(\gamma)} \tag{2.6}$$

Where $\zeta(\gamma)$ is the Riemann-zeta function. Hence for $k > 0$ power-law for discrete formalism is given as

$$\mathcal{P}_k = \frac{k^{-\gamma}}{\zeta(\gamma)} \tag{2.7}$$

Hence, the graph's degree distribution that follows a power law is called a scale-free graph.

2.1.2 Centralities

Centrality is an important tool for understanding graphs. This is used to calculate the significance of a given node in any graph. It gives a measure to rank nodes as per their properties. It is a scalar quantity assigned to each node in a graph to define the importance of each node based on the supposition. Detecting intermediate or central nodes is important as it affects the topology of the graph. For example, in a social network graph, the central position in a way represents popularity, high reputation, more influence, or leadership. Similarly, it can help to determine the molecules in a biological graph that can have an important biological role in signal transduction or detection of nodes that are highly interactive with other proteins [Pav+11].

Degree Centrality

The total number of edges connected to a vertex gives a degree of centrality. For a node i, the degree centrality can be defined as $\mathcal{C}_d(\mathcal{N}_i) = deg(i)$. For directed graphs, a node has an in-degree $\mathcal{C}_{d,in}(\mathcal{N}_i) = deg_{in}(i)$ and out-degree $\mathcal{C}_{d,out}(\mathcal{N}_i) = deg_{out}(i)$. The nodes with a high degree centrality are referred to as hubs as they are more central with large interactions. Scale-free graphs contain more hubs and have a high impact on the topology of the graph [Pav+11]. The fundamental formula for degree centrality $\mathcal{C}_d$ is given as [ZL17]

$$\mathcal{C}_d(\mathcal{N}_i) = \sum_{j=1}^{n} \mathcal{E}_{i,j(i\neq j)} \tag{2.8}$$

Betweenness Centrality

Betweenness centrality gives a measure to a node that plays an intermediate role in a graph. This node acts as a bridge between two neighboring nodes for communication. Thus it represents the most important nodes that are situated on a high proportion of paths between two groups of nodes in a graph. For two nodes $i, j, w \in \mathcal{V}(\mathcal{G})$, let $\sigma_{i,j}$ be a number of shortest paths between i to j. The number of shortest paths from i to j passing through w is denoted as $\sigma_{i,j}(w)$. Given $w \in \mathcal{V}(\mathcal{G})$, let $\mathcal{V}(i)$ represent all pairs of ordered set (i, j) in $\mathcal{V}(\mathcal{G}) \times \mathcal{V}(\mathcal{G})$ where i, j, w are distinct [Pav+11]. Hence betweenness centrality can be mathematically defined as

$$\mathcal{C}_b(w) = \sum_{(i,j)\in\mathcal{V}(w)} \frac{\sigma_{ij}(w)}{\sigma_{ij}} \tag{2.9}$$

Closeness Centrality

The nodes that can connect with other nodes with ease and convenience indicate closeness centrality, i.e. how close one node is to all the existing nodes in a graph. Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ is an undirected graph. The closeness centrality is defined as

$$\mathcal{C}_{close}(i) = \frac{1}{\sum_{i\in\mathcal{V}}^{|\mathcal{V}|} dist(i,j)} \tag{2.10}$$

Here, $dist(i,j)$ represents shortest path between i and j. It has been highly applicable in identifying the important central nodes or metabolites in genome-based metabolic graphs. As the closeness centrality for a graph or genome-based graph decreases, there has been an increase in the distance between paths. It has been the most important measure to find a metabolic node of graph or genome-based graph[Pav+11].

2.2 Neural Networks

Here we focus on the fundamental methods of neural networks. The first section (2.2.1) explains the mathematical model of the perceptron, whose goal is to approximate a model such as a classifier that maps input x to an output or class y. As neural networks involve optimization to improve their performance, (2.2.2) describes gradient-based optimization of model parameters for neural networks. As the forward propagation completes, backpropagation allows the cost to flow backward through the network and recompute the gradient to improve network performance. Hence the mathematical formulations for backpropagation are depicted in the section (2.2.3).

2.2.1 Feed-forward Network

The regression and classification problems are based on nonlinear basis functions $\phi_j(x)$ that are linearly combined and mathematically defined as,

$$y(\mathbf{x}, \mathbf{w}) = f\left(\sum_{j-1}^{M} w_j \phi_j(\mathbf{x})\right) \tag{2.11}$$

Where $f(\cdot)$ is a nonlinear activation function. The goal is to create a model in which the basis function $\phi_j(x)$ relies on parameters. The model adjusts the parameters and coefficients $\{w_j\}$ during training. The series of functional transformations of nonlinear function (Equation 2.11) gives the neural network model. Let input variables $x_1, ..., x_D$ be applied for M linear combinations,

$$a_j = \sum_{i=1}^{D} w_{ji}^{(1)} x_i + w_{j0}^{(1)} \tag{2.12}$$

where $j = 1, ..., M$, (1) in superscript denotes the first layer parameters of the neural network. Here $w_{j1}^{(1)}$) is the *weights* parameters, $w_{j0}^{(1)}$ is the *biases* parameters and a_j is termed as *activations*. Finally (Equation 2.12) is transformed using nonlinear *activation function* $h(\cdot)$ given as

$$z_j = h(a_j) \tag{2.13}$$

This value is the output of the basis function (Equation 2.11), also called *hidden units*. The nonlinear functions $h(\cdot)$ generally are logistic sigmoid or the tanh function. (Equation 2.13) is combined linearly to generate *unit activations*.

$$a_k = \sum_{j=1}^{M} w_{kj}^{(2)} z_j + w_{k0}^{(2)} \tag{2.14}$$

where $k = 1, ..., K$, and K is the number of outputs. (Equation 2.14) represents the transformation of the second layer neural network, and $w_{k0}^{(2)}$ are bias parameters. An appropriate activation function is chosen as per the nature of the data and the output unit activations as in (Equation 2.14) are transformed to get network outputs y_k.

The activation function for a standard regression problem identifies the function, such that $y_k = a_k$. The output unit activation for multiple binary classification tasks is transformed by a logistic sigmoid function hence,

$$y_k = \sigma(a_k) \tag{2.15}$$

where,

$$\sigma(a) = \frac{1}{1 + exp(-a)} \tag{2.16}$$

Similarly, for multiclass classification tasks, a softmax activation function is applied and is expressed as

$$p(\mathcal{C}_k|\mathbf{x}) = \frac{p(\mathbf{x}|\mathcal{C}_k)p(\mathcal{C}_k)}{\sum_k p(\mathbf{x}|\mathcal{C}_j)p(\mathcal{C}_j)} \tag{2.17}$$

$$= \frac{\mathbf{exp}(a_k)}{\sum_j \mathbf{exp}(a_j)} \tag{2.18}$$

It is a *normalized exponential* and is considered as a generalization of logistic sigmoid for multiclass problems. Finally, all the above stages are combined and given a sigmoidal output unit activation function expressed as

$$y_k(\mathbf{x}, \mathbf{w}) = \sigma\left(\sum_{j=1}^{M} w_{kj}^{(2)} h + \left(\sum_{i=1}^{D} w_{ji}^{(1)} x_i + w_{j0}^{(1)}\right) + w_{k0}^{(2)}\right) \tag{2.19}$$

Where vector $\mathbf{w}$ is combined with weight and bias parameters. Hence, the neural network is a nonlinear function of input variables $\{x_i\}$ that output variables $\{y_k\}$ and are supervised with vector $\mathbf{w}$. The (Equation 2.19) can be represented as forward propagation in the neural network as depicted in (Figure 2.1).

The bias parameters in (Equation 2.12) are merged into weight parameters with an input variable x_0. Here x_0 is clamped at 1 and the equation takes the form

$$a_j = \sum_{i=0}^{D} w_{ji}^{(1)} x_i \tag{2.20}$$

Similarly, the final network function can be defined by absorbing the second-layer biases into the second-layer weights.

$$y_k(\mathbf{x}, \mathbf{w}) = \sigma\left(\sum_{j=1}^{M} w_{kj}^{(2)} h + \left(\sum_{i=1}^{D} w_{ji}^{(1)} x_i\right)\right) \tag{2.21}$$

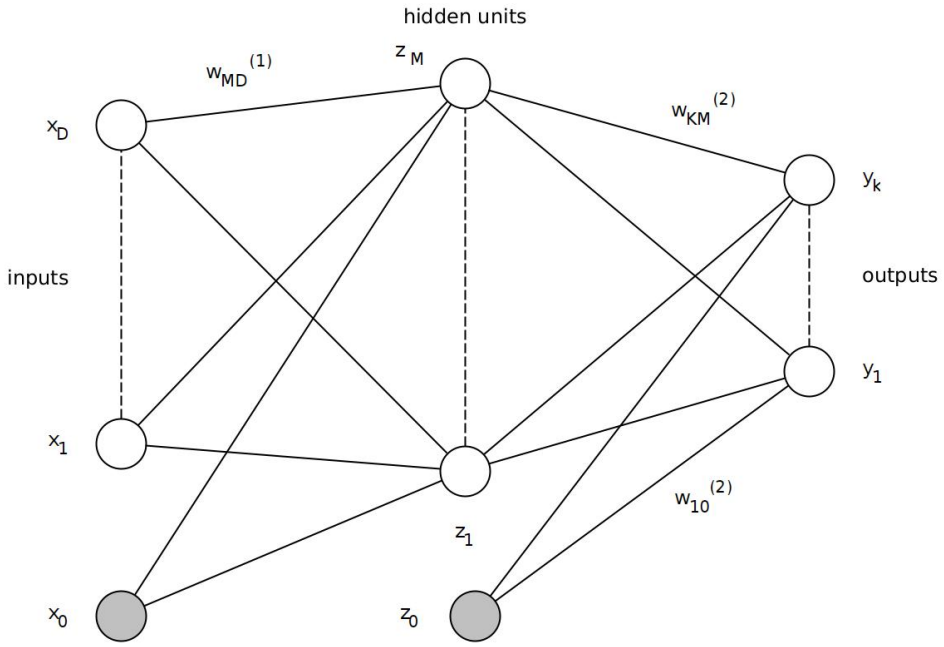

Figure 2.1: A two-layer neural network corresponding to (2.19). Nodes represent input, hidden, and output variables, and weight parameters are represented by links between the nodes, where the bias parameters are denoted by links from additional input x_0 and hidden variable z_0. The direction of information flow during forward propagation is shown by the arrow [BN06].

2.2.2 Gradient Descent Optimization

Gradient descent is an algorithm to perform the optimization of neural networks. A neural network can be a highly iterative process that consists of layers, inputs, objective functions, and weight parameters combined in an architecture. For a given architecture, the values of parameters determine how accurately the model performs the task, whereas the loss function determines the performance of the model. The target is to minimize the loss as a result, the best possible parameters are identified that match prediction with reality. Hence to optimize the model parameters of the neural network, Gradient Descent is used [Rud16].

The basic method to use gradient information is to update the weight $\mathbf{w}$ by taking a small step in the negative gradient direction as given in (Equation 2.22).

$$\mathbf{w}^{(\tau+1)} = \mathbf{w}^{(\tau)} - \eta \nabla E(\mathbf{w}^{(\tau)}) \tag{2.22}$$

where τ is the iteration step, $\eta > 0$ is the learning rate, $\nabla E(\mathbf{w}^{(\tau)})$ is rate of increase

of the error function. Since the error function is to be reduced, a step in the opposite direction is taken as -$\nabla E(\mathbf{w}^{(\tau)})$.

In each iteration, the gradient is evaluated to generate a new weight vector. There are variants of gradient descent, they differ as per the amount of data applied to find the gradient. The technique that calculates the gradient for all data sets at once is termed batch gradient descent. Such a method can be slow in processing and memory inefficient. However stochastic gradient descent updates the parameter for each data point at a time, so that

$$\mathbf{w}^{(\tau+1)} = \mathbf{w}^{(\tau)} - \eta \nabla E_n(\mathbf{w}^{(\tau)}) \tag{2.23}$$

Stochastic gradient descent is fast, as it performs one update at a time and can fluctuate the loss. The fluctuation can help the algorithm jump to a new and better minimum [Rud16].

2.2.3 Backpropagation

The main objective of the back-propagation algorithm is to evaluate the gradient of an error function $E(\mathbf{w})$ for a feed-forward neural network. Assuming a simple linear model having a linear combination of inputs x_i, which outputs y_k so that

$$y_k = \sum_i w_{ki} x_i \tag{2.24}$$

and the error function for a single input pattern n is defined as

$$E_n = \frac{1}{2} \sum_k (y_{nk} - t_{nk})^2 \tag{2.25}$$

where y_{nk} is the predicted vector, t_{nk} is the target vector, and $y_{nk} = y_k(\mathbf{x}_n, \mathbf{w})$. The error function is used to compute the gradient with respect to weight w_{ji} so that

$$\frac{\partial E_n}{\partial w_{ij}} = (y_{nj} - t_{nj}) x_{ni} \tag{2.26}$$

This simple gradient of the error function can be extended to more complex multi-layer feed-forward networks. In the feed-forward neural network, the weighted sum of inputs is computed by each unit and is given by

$$a_j = \sum_i w_{ji} z_i \tag{2.27}$$

where z_i is unit activation, that links unit j, and w_{ji} is the weight for that link. The biases are not dealt with explicitly as they can be included by an extra unit in the sum as in (Equation 2.11). A nonlinear activation function $h(\cdot)$ transforms the sum in (2.27) that outputs activation z_j of unit j so that

$$z_j = h(a_j) \tag{2.28}$$

During training, an input vector is supplied to the network, and activations for hidden and output network is calculated. This is called forward propagation. Next, the derivative of E_n with respect to weight w_{ji} is assessed. The input pattern n will regulate the outputs of each unit. Here the subscript n will not be considered to make the notation clear. A chain rule is applied to find partial derivatives of E_n which depends on the weight w_{ji} to give

$$\frac{\partial E_n}{\partial w_{ji}} = \frac{\partial E_n}{\partial a_j} \frac{\partial a_j}{\partial w_{ji}} \tag{2.29}$$

Let the partial derivatives be denoted as

$$\delta_j = \frac{\partial E_n}{\partial a_j} \tag{2.30}$$

From (Equation 2.27), we can write

$$\frac{\partial a_j}{\partial w_{ji}} = z_i \tag{2.31}$$

Substituting (Equation 2.30 and 2.31) into (Equation 2.29), we obtain

$$\frac{\partial E_n}{\partial w_{ji}} = \delta_j z_i \tag{2.32}$$

Here, to find the derivatives, computation of δ_j for each hidden and output unit is required and applied to (Equation2.32). The output unit is defined as

$$\delta_k = y_k - t_k \tag{2.33}$$

Finally, to compute the $\delta' s$ for hidden units, the chain rule is applied for partial derivatives, such that

$$\delta_j = \frac{\delta E_n}{\partial a_j} = \sum_k \frac{\partial E_n}{\partial a_k} \frac{\partial a_k}{\partial a_j} \tag{2.34}$$

Hence, we obtain the backpropagation formula by substituting the δ given in (Equation 2.30) into (Equation 2.34) and using (Equation 2.27 and 2.28). During the process of backpropagation, the weight of the network is updated for future predictions.

$$\delta_j = h^{'}(a_j) \sum_k w_{kj} \delta_k \tag{2.35}$$

2.3 Graph Machine Learning

The real world has various objects and connections to each other. These objects and their relational properties are naturally expressed as graphs. Graph neural networks (GNNs) are a kind of neural network that generates an embedding for a graph that represents the input graph. Different kinds of data from physical systems, molecular fingerprints, and protein interfaces are represented in graphs as they have the great expressive power to hold the relational information. Graph neural networks learn such complex structures of graphs via message passing. Recent developments in graph neural networks such as graph recurrent network (GRN), graph convolutional network (GCN), and graph attention network (GAT) have given exceptional performances on graph data [Zho+20].

2.3.1 Applications and Tasks of Graph ML

Graph neural networks are applied to different problems such as protein folding, recommender systems, drug discovery, traffic prediction, molecule generation or optimization, physical simulations, etc. Some of the important tasks performed on the

above-mentioned problems are node classification, link prediction, graph classification, clustering, graph generation, and graph evolution, some of which are described below [Ham20].

Graph Classification

In a graph classification task, a set of graph data is learned by a model and the goal is to predict specific graph-level properties. Variants of GNNs, such as graph convolutional networks (GCN) have high discriminative power for classification tasks. Each layer of GCN aggregates node representations and edges to construct graph-level representations for the entire graph. Molecule property prediction can be considered as a graph classification problem, considering molecules as graph data and its property as graph labels [Xu+18].

Graph Generation

Generating graphs is the basis for exploring physical phenomena, social sciences, and biological networks. Generative models for graphs are a powerful method to learn the attributes and structures of graphs, that apply graph neural networks for learning and generation or extension of graphs [Li+18]. DGMG [Li+18] and GraphRNN [You+18] are such methods based on a deep auto-regressive model that generates graphs by learning from a set of input graph data represented as a sequence of nodes and edges [You+18].

2.3.2 Graph Neural Networks

Graph neural networks (GNN) are a method to define deep neural networks on graph data. The idea is to generate representations of nodes and feature information that represent the actual structure of the graph. GNN uses neural message passing in-order to exchange vector messages between nodes and update using deep neural networks.

Overview of Message Passing

The key idea is to generate node embedding based on local network neighborhoods. In each iteration of a message passing in a graph neural network, an embedding $\mathbf{h}_u^{(k)}$ is generated for every node $u \in \mathcal{V}$, with the vector message aggregated from neighborhood nodes $\mathcal{N}(u)$ of u (2.2).

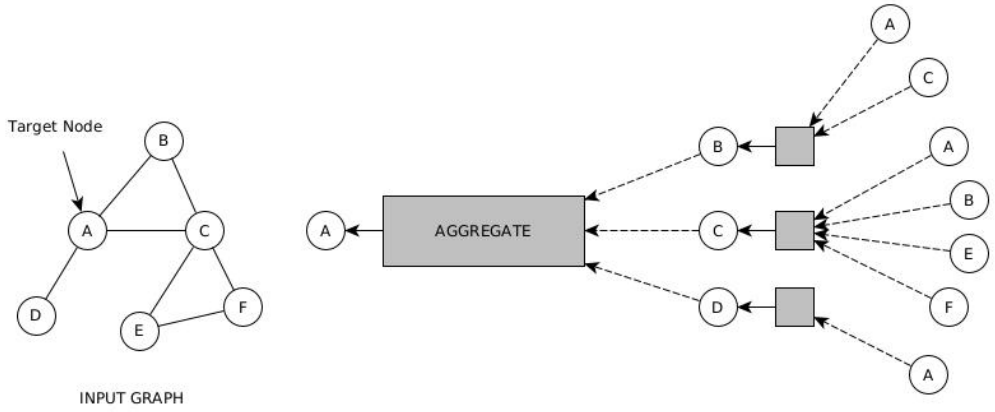

Figure 2.2: Shows an aggregation of messages to a single node from its local neighbor- hood. The model sums up messages to A with its local graph neighbors (i.e., B, C, and D), and these vector messages from neighbors are in turn aggregated from their corresponding neighborhoods. The figure depicts message-passing in two-layer where the tree structure is formed during graph computation with GNN [Ham20].

Figure 2.2, shows an aggregation of messages to a single node from its local neighborhood. The model sums up messages to A with its local graph neighbors (i.e., B, C, and D), and these vector messages from neighbors are in turn aggregated from their corresponding neighborhoods. The figure depicts message-passing in two-layer where a tree structure is formed during graph computation with GNN. This message-passing can be mathematically expressed as follows:

$$\mathbf{h}_u^{(k+1)} = \mathbf{UPDATE}^{(k)}\left(\mathbf{h}_u^{(k)}, \mathbf{AGGREGATE}^{(k)}(\{\mathbf{h}_v^k, \forall v \in \mathcal{N}(u)\})\right) \tag{2.36}$$

$$\mathbf{h}_u^{(k+1)} = \mathbf{UPDATE}\left(\mathbf{h}_u^{(k)}, \mathbf{m}_{\mathcal{N}(u)}^{(k)}\right) \tag{2.37}$$

Here, **UPDATE** and **AGGREGATE** are neural networks, and $\mathbf{m}_{\mathcal{N}(u)}$ is message vector aggregated from neighbor $\mathcal{N}_{(u)}$ of u. In each k iteration of the graph neural network, a message $\mathbf{m}_{\mathcal{N}(u)}^{(k)}$ is generated by aggregating the node embeddings of $u's$ neighborhood $\mathcal{N}(u)$. The **UPDATE** function combines previous embeddings $\mathbf{h}_u^{(k-1)}$ of node u with the message $\mathbf{m}_{\mathcal{N}(u)}^{(k)}$, to generate the final embedding $\mathbf{h}_u^{(k)}$. The embeddings at $k = 0$ represent the features of input nodes, i.e., $\mathbf{h}_u^{(0)} = \mathbf{x}_u, \forall u \in \mathcal{V}$. Completing K iteration of message passing in a graph neural network, the embeddings for each node are the output of the end layer, i.e.,

$$\mathbf{z}_u = \mathbf{h}_u^{(K)}, \forall u \in \mathcal{V} \tag{2.38}$$

These embeddings encode information about the graph in two basic forms. One is *structural information* about the graph, which might represent information about the degrees of all the nodes in a graph. This encoding is useful in analyzing molecular graphs, predicting atom types, etc. On the other hand, GNN captures *feature-based* information by aggregating information based on the local neighborhood.

Graph Pooling

The neural message passing generates node representations $\mathbf{z}_u, \forall v \in \mathcal{V}$, in the form of embeddings. But in the case of graph-level predictions, learning embedding $\mathbf{z}_\mathcal{G}$ is important for the entire graph $\mathcal{G}$. The goal is to combine the node embeddings together and learn embedding for the whole graph, termed as *graph pooling*. A pool function f_p is designed to map node embedding $\{\mathbf{z}_1, ..., \mathbf{z}_{|V|}\}$ to generate an new embedding $\mathbf{z}_\mathcal{G}$ representing entire graph. A simple approach for learning graph-level embeddings is taking the sum of all node embeddings (Equation 2.39):

$$\mathbf{z}_\mathcal{G} = \frac{\sum_{v \in \mathcal{V}} \mathbf{z}_u}{f_n(|\mathcal{V}|)} \tag{2.39}$$

Here, the function f_n is used for normalization, and the method of mean-based pooling of node embeddings is generally applicable for small graphs.

Generalized Message Passing

Previously discussed message-passing is designed for node-level operation. However, the generalization of message-passing for edge and graph-level information is equally important. The following equations show generalized message-passing:

$$\mathbf{h}_{(u,v)}^{(k)} = \mathbf{UPDATE}_{edge}\left(\mathbf{h}_{(u,v)}^{(k-1)}, \mathbf{h}_{(u)}^{(k-1)}, \mathbf{h}_{\mathcal{G}}^{(k-1)}\right) \tag{2.40}$$

$$\mathbf{m}_{\mathcal{N}(u)} = \mathbf{AGGREGATE}_{node}\left(\mathbf{h}_{(u,v)}^{(k)}, \forall v \in \mathcal{N}(u)\right) \tag{2.41}$$

$$\mathbf{h}_{u}^{(k)} = \mathbf{UPDATE}_{node}\left(\mathbf{h}_{u}^{(k-1)}, \mathbf{m}_{\mathcal{N}(u)}, \mathbf{h}_{\mathcal{G}}^{(k-1)}\right) \tag{2.42}$$

$$\mathbf{h}_{\mathcal{G}}^{(k)} = \mathbf{UPDATE}_{graph}\left(\mathbf{h}_{\mathcal{G}}^{(k-1)}, \{\mathbf{h}_{u}^{(k)}, \forall u \in \mathcal{V}\}, \{\mathbf{h}_{(u,v)}^{(k)} \forall (u,v)) \in \mathcal{E}\}\right) \tag{2.43}$$

In the case of generalized message-passing, hidden embeddings $\mathbf{h}_{(u,v)}^{(k)}$ for each edge and the entire graph embeddings $\mathbf{h}_{\mathcal{G}}^{(k)}$ are generated. As a result, the edge and graph level information is integrated into the embeddings with the message-passing model. In terms of generalized message-passing, edge embeddings are updated based on incident node embeddings (Equation 2.40). Secondly, the node embeddings are updated by aggregating all incident edge embeddings (Equation 2.41 and 2.42). Finally, graph-level embedding is an aggregation of node and edge embeddings (Equation 2.43).

2.3.3 The Basic GNN

GNN is basically an iteration of message-passing with **UPDATE** and **AGGREGATE** and these functions are building blocks to design the basic graph neural network defined as

$$\mathbf{h}_{u}^{(k)} = \sigma\left(\mathbf{W}_{self}^{(k)}\mathbf{h}_{u}^{(k-1)} + \mathbf{W}_{neigh}^{(k)} \sum_{v \in \mathcal{N}(u)} \mathbf{h}_{v}^{k-1} + \mathbf{b}^{(k)}\right) \tag{2.44}$$

Here, $\mathbf{W}_{self}^{(k)}$, $\mathbf{W}_{neigh}^{(k)} \in \mathbb{R}^{d^{(k)} \times d^{(k-1)}}$ are parameters to be trained and σ is an element-wise non-linear function. It can be either a *tanh*, *softmax*, or *ReLU* function as per the need of an operation. Bias term $\mathbf{b}^{(k)} \in \mathbb{R}^{d^{(k)}}$ helps to increase the performance. The Equation (2.44) is parallel to multi-layer perceptron, where linear and element-wise non-linear operations are performed.

GNN can be defined through **UPDATE** and **AGGREGATE** functions:

$$\mathbf{m}_{\mathcal{N}_{(u)}} = \sum_{v \in \mathcal{N}_{(u)}} \mathbf{h}_v, \tag{2.45}$$

$$\mathbf{UPDATE}(\mathbf{h}_u, \mathbf{m}_{\mathcal{N}_{(u)}}) = \sigma\left(\mathbf{W}_{self}\mathbf{h}_u + \mathbf{W}_{neigh}\mathbf{m}_{\mathcal{N}(u)}\right) \tag{2.46}$$

The message aggregated from the neighborhood of $u's$ graph can be expressed as,

$$\mathbf{m}_{\mathcal{N}(u)} = \mathbf{AGGREGATE}^{(k)}\left(\mathbf{h}_v^{(k)}, \forall v \in \mathcal{N}(u)\right) \tag{2.47}$$

Message Passing with Self-loops

To simplify message passing, the update method is omitted and a self-loop is applied to the input graph. Equation(2.48) defines message passing as self-loop.

$$\mathbf{h}_u^{(k)} = \mathbf{AGGREGATE}\left(\{\mathbf{h}_v^{(k-1)}, \forall v \in \mathcal{N}(u) \cup \{u\}\}\right) \tag{2.48}$$

Here the definition of the UPDATE function is no longer required, as it is done through aggregating the set $\mathcal{N}(u) \cup \{u\}$, i.e., set of node neighbors and set of the node itself. This approach reduces overfitting but restricts the expressivity of the graph neural network, as the embeddings from the node and its neighbors cannot be differentiated.

Neighborhood Normalization

The basic approach of aggregation sums up the embeddings from neighbors, which can be unstable and difficult for optimization. One simple method to solve this problem is the normalization of the aggregated embeddings i.e., take an average of embeddings instead of the sum as in Equation(2.49), or *symmetric normalization* as in Equation(2.50):

$$\mathbf{m}_{\mathcal{N}_{(u)}} = \frac{\sum_{v \in \mathcal{N}_{(u)}} \mathbf{h}_v}{\mathcal{N}(u)}, \tag{2.49}$$

$$\mathbf{m}_{\mathcal{N}_{(u)}} = \sum_{v \in \mathcal{N}(u)} \frac{\mathbf{h}_v}{\sqrt{|\mathcal{N}(u)||\mathcal{N}(v)|}}. \tag{2.50}$$

2.3.4 Graph Convolution Networks (GCNs)

Graph convolutional network (GCN)- is the most widely used standard graph neural network based on symmetric normalization (Equation 2.50) and self-loop update (Equation 2.48) methods. Message-passing for GCN is defined as

$$\mathbf{h}_u^{(k)} = \sigma\left(\mathbf{W}^{(k)} \sum_{v \in \mathcal{N}(u) \cup \{u\}} \frac{\mathbf{h}_v}{\sqrt{|\mathcal{N}(u)||\mathcal{N}(v)|}}\right). \tag{2.51}$$

The final embeddings are used to make predictions

$$\hat{y_u} = \mathbf{PREDICT}(\mathbf{h}_u^{(K)}) \tag{2.52}$$

Here **PREDICT** is a neural network, learned by the GCN model. For every step k, σ, $\mathbf{W}^{(k)}$ are shared throughout the nodes.

2.3.5 Evaluation Metrics

The machine learning models are designed to solve problems, and it is crucial to determine the performance of the model. Some commonly used metrics are accuracy, and maximum mean discrepancy (MMD). Depending on the machine learning model, metrics are taken into consideration. Accuracy is reliable to evaluate a classifier when the categories of the datasets are balanced. Additionally, the MMD score would be a meaningful choice for evaluating a generative model.

Accuracy

To determine the performance of the machine learning model's prediction, the simplest evaluation metric is accuracy. It is the ratio of correctly classified predictions to the total number of predictions made. Accuracy is a commonly used metric for classification models, where the objective is to predict the class of input given to the machine learning model. Mathematically, accuracy is defined as in (2.53), where Te represents the test datasets and $I[\cdot]$ is an indicator function. It indicates 1 if the argument evaluates to true, and 0 otherwise. Here, $\hat{p}(x)$ is the estimated class, and $p(x)$ is the true class [Fla12].

$$Accuracy = \frac{1}{|Te|} \sum_{x \in Te} I[\hat{p}(x) = p(x)] \tag{2.53}$$

Maximum Mean Discrepency

Maximum mean discrepancy (MMD) is a kernel-based statistical test [TSS16]. The statistical similarity of two given distributions can be determined with MMD. Generative models utilize it to measure its performance by assessing the distance metric between two sets of samples. The deep generative model in this research is used to compare the distribution of training data and data generated by the generative model, to evaluate the generative model performance. Mathematically MMD between two distributions $\mathcal{P}$ and $\mathcal{Q}$ can be written as,

$$MMD(\mathcal{P}, \mathcal{Q}) = \| \mu_{\mathcal{P}} - \mu_{\mathcal{Q}} \|_{\mathcal{F}} \tag{2.54}$$

Where $\mathcal{F}$ is a reproducing kernel Hilbert space. One common choice of kernel function for the experiment would be the Gaussian kernel $k(x_i, y_j)$.

$$k(x_i, y_j) = exp\left(\frac{- \parallel x_i - y_j \parallel^2}{2\sigma^2}\right) \tag{2.55}$$

2.4 Finite-State Machine

A Finite-state machine is a computational model with a set of *states* the machine can be in. The change of states from one to another is called *transition* and the machine transits between states to perform different actions. Finite-state machines can be categorized into deterministic finite-state machines (DFA) and non-deterministic finite-state machines (NFA). The term "deterministic" indicates, on each input to a current state, out of several states, there can be only one state to which the machine can transition from its current state. On the other hand, "non-deterministic" has the power to be in several possible states for a given input. Here the focus would be more on the non-deterministic finite-state machines.

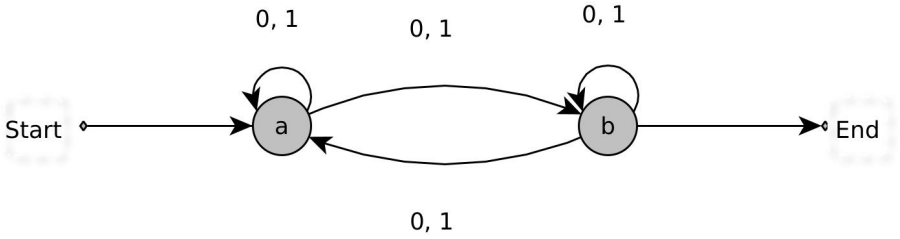

Figure 2.3: Shows non-deterministic finite-state machine, that accepts input 0 or 1 and can transition to any existing state from the current state. Here, the state a can take input as 0 and can transit to either state a or b.

A non-deterministic finite-state machine (NFA) is represented as 2.56 where,

- $\mathcal{Q}$ is a finite set of *states*. Figure 2.3 shows two states $\mathcal{Q} = \{a, b\}$.
- Σ is a finite set of input symbols given as $\Sigma = \{0, 1\}$.
- q_0 is one of the state of $\mathcal{Q}$. It is the start state.
- $\mathcal{F}$ is a set of final states and is a subset of $\mathcal{Q}$.

- δ is the transition function, that takes a state in $\mathcal{Q}$ and one of input symbol Σ. It can return multiple states that a machine can transition from the current state.

$$\mathcal{A} = \left(\mathcal{Q}, \Sigma, \delta, q_0, \mathcal{F} \right) \tag{2.56}$$

3 Methods

3.1 Generation of Synthetic Graph Datasets For Training and Evaluation

The experiments consider sets of synthetic graph datasets. Three different models of synthetic undirected graphs, the Erdos-Renyi, Barabasi-Albert, and Watts-Strogatz models are generated. Firstly, the datasets are created and stored permanently to ensure consistency in the data and the experiments. To introduce diversity in the experiment and empirical research, graphs from the probabilistic model to uniformly generated random graphs are taken into consideration. These graph datasets are used to conduct experiments on empirical research, classification problems, and generative models. Below is a discussion of each type of graph generation method, along with information about synthetic datasets for each graph type.

Erdos Renyi Graph :

It is a random graph model where a graph is constructed by connecting vertices randomly. Let $\mathcal{G}_{(n,p)}$ be a model in which a graph has n vertices and each edge between the vertices is linked with probability p. As a result, the probability of generating a graph through the model with n vertices and M edges is given as follows:

$$p^M(1-p)^{(\frac{1}{2})-M} \tag{3.1}$$

The graph dataset sample has vertices $|v| = 83$ and is randomly connected with a probability of $p = 0.1$.

Barabasi-Albert Graph :

A common property of real-world graphs is that they follow a scale-free power-law degree distribution. Barabasi-Albert also known as the BA model is a consequence of graph expansion by adding new vertices and preferential attachment. In preferential attachment, a probabilistic method is applied where a newly created vertex is free to join any other vertex in the graph. The graph starts with m_0 vertex, and the edges are selected at random such that each vertex has a minimum of one edge. The graph evolves in two steps: **growth** and **preferential attachment**. In each step, a new vertex with $m(\leq m_0)$ edges joins the new vertex to the existing graph. Secondly, the probability of joining the new vertex to the existing vertices i in a graph depends on the degree k_i given as [BA99]

$$\Pi(k_i) = \frac{k_i}{\sum_j k_j} \tag{3.2}$$

Barabasi-Albert graphs are generated with vertex $|v| = 83$. During generation, each node in the graph is connected to three existing nodes.

Watts-Strogatz Graph :

It is a random graph generation model that generates small-world graphs in which a regular graph is rewired to introduce disorder in the graph. These graphs are highly clustered and have small path lengths. The rewiring procedure interpolates between a regular graph and a random graph where the number of edges and vertices are unchanged. It starts with a regular ring lattice of n vertices and m edges per vertex. All vertices are linked to their k nearest neighbors with undirected edges. Each edge is rewired randomly with a probability p. Thus the procedure tunes the graph between $p = 0$ and $p = 1$, i.e. from a regular lattice ring to a randomly wired graph [WS98].

Watts-Strogatz graphs with vertex $|v| = 83$ are generated. Each node is set to join $k = 7$ nearest neighbors in a ring topology with a probability of $p = 0.7$ to rewire the edges.

3.2 Operations for Graph Manipulation

The existing graph models discussed in section 3.1 follow two basic steps in the process of evolving a graph as per the input parameter: adding a vertex (f_{add_node}) and connecting pair of vertices with an edge (f_{add_edge}). These steps can be considered as two states of a state machine diagram, where the respective graph model iteratively switches between these two states as per the model's algorithm and generates the graph. Technically these states are named **Decision Operations**.

Apart from using a graph model algorithm to generate a graph, generative models in deep learning such as DGMG [Li+18], DeepGG [SG21], and GraphRNN [You+18] are capable of learning the attributes and structure of graphs. Hence, the learned generative model can construct graphs that are structurally similar to the learned graphs without using any specific graph algorithm. The existing deep generative models transit between the basic two operations f_{add_node} and f_{add_edge} to construct a graph. One of the major efforts of this research is to design more complex local operations and extend the state machine model into more states, which was previously limited to two simple states. Section 3.2.1 explains the design of complex decision operations that would be considered as multiple states of the extended state machine model (3.9).

3.2.1 Decision Operations for Graph Manipulation

As discussed before in section (3.2) about DGMG, and DeepGG, the graph generation process is modeled in a state machine having three states namely *addnode*, *addedge*, and *selectnode*, for graph expansion [Li+18] [SG21]. To extend the idea of a deep-state machine with more than two states, nine varieties of simple to complex decision operations are designed. Most of the operations are designed to be very close to the characteristics and properties of system biology. Here is a list of decision operations modeled for experiments on extending the state machine with many states for the graph generation.

Adding one vertex to existing input graph or empty input graph :

Any graph starts with a vertex, which makes it a crucial decision that can initiate the graph construction process or add a new vertex in between the construction. Algorithm 1, shows the procedure for adding a new vertex. The algorithm takes

graph $\mathcal{G}$ as the input parameter with n vertices, adds a new vertex, and returns a new graph with $n + 1$ vertices.

Algorithm 1: `Add a new vertex`

Input: Undirected graph $\mathtt{G_{in}} = (\mathtt{V_n}, \mathtt{E_m})$
Output: Undirected graph $\mathtt{G_{out}} = (\mathtt{V_{n+1}}, \mathtt{E_m})$
Function `add_node(`*input_graph: G_{in}*`):`

```
G_new = copy(G_in)
new_vertex = max(G_new.nodes) + 1
G_new.add_node(new_vertex)
return G_new (= G_out)
```

Adding one vertex and an edge between input graph and the new vertex:

Another important decision is to connect pair of vertices. It generates an edge between the newly created vertex and any vertex from the existing input graph. Algorithm 2, depicts the addition of an edge between pair of vertices. The algorithm takes graph $\mathcal{G}$ as the input parameter with n vertices and m edges, adds a new vertex, and connects it to a random vertex of the input graph. It returns a new graph with $n + 1$ vertices and $m + 1$ edges.

Algorithm 2: `Add a new vertex and edge`

Input: Undirected graph $\mathtt{G}_{in} = (\mathtt{V}_n, \mathtt{E}_m)$
Output: Undirected graph $\mathtt{G}_{out} = (\mathtt{V}_{n+1}, \mathtt{E}_{m+1})$
Function `add_edge(`*input_graph: G_{in}*`):`

```
G_new = copy(G_in)
new_vertex = max(G_new.nodes)
random_vertex = choice(list(G_new))
G_new.add_node(new_vertex)
G_new.add_edge(new_vertex, random_vertex)
return G_new (= G_out)
```

Adds a vertex and connects it to the highest degree centrality vertex of input graph:

In biological networks, central vertices are crucial, and they have an impact on the network topology. A vertex with a high degree centrality is highly connected with

large interactions [Pav+11]. The operation adds a vertex and connects it to a vertex with the highest degree centrality from the given input graph $\mathcal{G}$, as in Algorithm 3. It returns a new graph with $n+1$ vertices and $m+1$ edges, where n, m is the number of vertices and edges of an input graph respectively. This gives the flexibility to expand a graph with more complexity since many domains of the graph incorporate the characteristics of high degree centrality vertex.

Algorithm 3: `Adds edge between new vertex and vertex with high degree centrality of input graph`

Input: Undirected graph $\mathtt{G_{in}} = (\mathtt{V_n}, \mathtt{E_m})$
Output: Undirected graph $\mathtt{G_{out}} = (\mathtt{V_{n+1}}, \mathtt{E_{m+1}})$
Function `add_node_high_deg_cntra(`*input_graph: G_{in}*`)`:

```
G_new = copy(G_in)
high_deg_cntra = max(degree_centrality(G_new))
new_vertex = max(G_new.nodes)+ 1
G_new.add_node(new_vertex)
G_new.add_edge(new_vertex, high_deg_cntra)
return G_new (= G_out)
```

Adds a vertex and connects it to the highest closeness centrality vertex of input graph:

Closeness centrality is a measure to find out important vertices that can connect other vertices in a graph quickly. These characteristics are applied in recognition of top central metabolites in a genome-based network [Pav+11]. The operation is used to add a vertex and connect it to another vertex with the highest closeness centrality in a given input graph $\mathcal{G}$, as defined in Algorithm 4. It returns a new graph with $n+1$ vertices and $m+1$ edges, where n, m is the number of vertices and edges of an input graph respectively. Here vertex with label $n+1$ is connected to the highest closeness centrality vertex of the input graph.

Adds a vertex and connects it to the highest betweenness centrality vertex of input graph:

Vertices that lie on the pathway are extensively used to communicate between neighboring vertices and have high betweenness centrality. For example, protein metabolites govern the flux between two big metabolic modules [Pav+11]. The operation

Algorithm 4: Adds edge between new vertex and vertex with high closeness centrality of input graph

Input: Undirected graph $G_{in} = (V_n, E_m)$
Output: Undirected graph $G_{out} = (V_{n+1}, E_{m+1})$
Function add_node_high_close_cntra(*input_graph: G_{in}*):

```
G_new = copy(G_in)
high_close_cntra = max(closeness_centrality(G_new))
new_vertex = max(G_new.nodes)+ 1
G_new.add_node(new_vertex)
G_new.add_edge(new_vertex, high_close_cntra)
return G_new (= G_out)
```

is used to add a new vertex and connect it to a vertex with the highest betweenness centrality in a given input graph $\mathcal{G}$ as in Algorithm 5. The algorithm returns an output graph having $n + 1$ vertices and $m + 1$ edges, where n, m is the number of vertices and edges of a given input graph respectively. Here the highest betweenness centrality vertex is connected to the new vertex with the label $n + 1$.

Algorithm 5: Adds edge between new vertex and vertex with high betweenness centrality of input graph

Input: Undirected graph $G_{in} = (V_n, E_m)$
Output: Undirected graph $G_{out} = (V_{n+1}, E_{m+1})$
Function add_node_high_bwtn_cntra(*input_graph: G_{in}*):

```
G_new = copy(G_in)
high_bwtn_cntra = max(betweenness_centrality(G_new))
new_vertex = max(G_new.nodes)+ 1
G_new.add_node(new_vertex)
G_new.add_edge(new_vertex, high_bwtn_cntra)
return G_new (= G_out)
```

Adds a vertex and connects it to the highest eccentricity centrality vertex of input graph :

Eccentricity centrality provides an estimate of the accessibility of a vertex from other vertices. In biological networks, proteins with high eccentricity play an important role. They can easily adapt to changes in the density of molecules linked to them [Pav+11]. Based on this principle, the operation adds a vertex and connects it to a vertex with the highest eccentricity centrality in a given input graph $\mathcal{G}$, as shown in

Algorithm 6. Similar to previous algorithms, it takes input as graph $\mathcal{G}$ and returns an output graph with $n+1$ vertices and $m+1$ edges. The newly added vertex with label $n+1$ is connected with the highest eccentricity centrality vertex of the input graph $\mathcal{G}$.

Algorithm 6: `Adds edge between new vertex and vertex with high eccentricity centrality of input graph`

Input: Undirected graph $G_{in} = (V_n, E_m)$
Output: Undirected graph $G_{out} = (V_{n+1}, E_{m+1})$
Function `add_node_high_ecc_cntra(`*input_graph:* G_{in}`):`

```
G_new = copy(G_in)
high_ecc_cntra = max(eccentricity_centrality(G_new))
new_vertex = max(G_new.nodes)+ 1
G_new.add_node(new_vertex)
G_new.add_edge(new_vertex, high_ecc_cntra)
return G_new (= G_out)
```

Adds one vertex and connects it to a random vertex of the input graph:

The operation is very similar to that in Algorithm 2. It involves adding a new vertex and connecting it to a random vertex of the input graph. However, in this case, the random vertex is chosen from the top three maximum-degree vertices. Algorithm 7, depicts the addition of a vertex and a random edge. The algorithm takes graph $\mathcal{G}$ as the input parameter with n vertices and m edges. It adds a new vertex and connects it to a random vertex chosen among the maximum degrees vertices of the input graph. It returns a new graph $\mathcal{G}$ with $n+1$ vertices and $m+1$ edges.

Adds one triangle and m random edges from triangle to the given input graph :

A triangle itself is a graph. Adding a triangle to an existing graph can be a more complex operation than just adding a node or an edge. However, to achieve graph generation with fewer steps, complex operations like adding a triangle can be beneficial. Algorithm 8, takes graph $\mathcal{G}$ with n vertices and *tri_n_edges* edges as an input parameter. Here *tri_n_edges* refers to the maximum number of edges that are to be connected from the vertices of the triangle to the existing input graph. The vertex of the triangle is chosen randomly, and the vertex of the input graph is also

Algorithm 7: Add one vertex and random edge

```
Data: Undirected graph G_in = (V_n, E_m)
Result: Undirected graph = G_out
Function add_one_node_random_edge(input_graph:G_in):
    G_new = copy(G_in)
    if G_new.nodes < 1 then
        raise Exception('Sorry, nodes less than one')
    end
    if max(G_new.nodes) >= 5 then
        deg_view_dict = get_key_value(nx.degree(G_new))
        max_deg_nodes = sorted(deg_view_dict)[:3]
        selected_node = random.choice(max_deg_nodes)
    else
        deg_view_dict = get_key_value(nx.degree(G_new))
        max_deg_nodes = sorted(deg_view_dict)[:1]
        selected_node = random.choice(max_deg_nodes)
    end
    v_new = max(G_new.nodes + 1)
    G_new.add_node(v_new)
    G_new.add_edge(v_new, selected_node)
    return G_new (= G_out)
```

chosen at random, but among the top five maximum-degree vertices. The pair of vertices obtained are joined with an edge. If an input graph $\mathcal{G}$ has only one vertex, in such cases the maximum number of edges between the triangle and input graph $\mathcal{G}$ will be three. The idea of choosing the vertex at random gives graph generation a non-deterministic nature.

Adds n vertices and m edges between input graph and new vertices:

Adding multiple vertices and edges in a single step during graph generation is more complex than working with a single vertex or edge. The approach also helps generate complex structures in an existing graph with fewer steps. Algorithm 9, takes graph $\mathcal{G}$, number of vertices n_nodes and number of edges m_edges as an input parameter. Here n_nodes, m_edges refers to the number of vertices and the maximum number of edges for each new vertex to be added for graph expansion. The vertex of the input graph $\mathcal{G}$ is chosen at random among the top five maximum degree vertices. The pair of vertices obtained are joined with an edge.

Algorithm 8: Add one triangle n edges

```
Data: Undirected graph G_in = (V_n, E_m)
Result: Undirected graph = G_out
Function add_one_tirangle_n_edges(input_graph:G_in , tri_n_edges=5):
    G_new = copy(G_in)
    if G_new.nodes < 1 then
        raise Exception('Sorry, nodes less than one')
    end
    deg_view_dict = get_key_value(nx.degree(G_new))
    max_deg_nodes = sorted(deg_view_dict)[:5]
    v_max =  max(G_new.nodes)
    G_new.add_edge(v_max+1, v_max+2)
    G_new.add_edge(v_max+2, v_max+3)
    G_new.add_edge(v_max+1, v_max+3)
    v_new = [v_max+1, v_max+2, v_max+3]
    for i in range(tri_n_egdes):
        tri_rand_vertex = random.choice(v_new)
        graph_max_deg_vertex = random.choice(max_deg_nodes)
        if not G_new.has_egde(tri_rand_vertex, graph_max_deg_vertex)
         then
            G_new.add_edge(tri_rand_vertex, graph_max_deg_vertex)
        end
    return G_new (= G_out)
```

3.2.2 Inverse Decision Operations for Graph Manipulation

In section (3.2.1), it was mentioned decision operations are applied for graph manipulation to expand the graph. However, understanding how a given complete graph was created is crucial. To achieve this, operations for backward processing are necessary. Therefore, inverse operations that are reciprocal to the decision operations are developed, which facilitate the backward path to track down the graph construction process. The subsequent operations are created and discussed as inverse decision operations for graph manipulation.

Remove a vertex from given input graph:

Removing a vertex is considered the inverse of adding a vertex. A very simple approach to deconstructing a graph is by removing a vertex. Algorithm 10, is modeled such that, it acts inversely to add a vertex as in Algorithm 1. It takes graph $\mathcal{G}$ as the input parameter with n vertices and m edges, removes a minimum degree vertex, and returns a new graph with $n-1$ vertices. Since Algorithm 1, adds

Algorithm 9: Add n nodes m edges

```
Data: Undirected graph G_in =(V_n, E_m)
Result: Undirected graph = G_out
Function add_n_nodes_m_edges(input_graph: , n_nodes=3, m_edges=3):
    G_new = copy(G_in)
    if G_new.nodes < 1 then
        raise Exception('Sorry, nodes less than one')
    end
    deg_view_dict = get_key_value(nx.degree(G_new))
    max_deg_nodes = sorted(deg_view_dict)[:5]
    v_max = max(G_new.nodes)
    v_new = []
    for n in range(n_nodes):
        v_new.append(v_max+1+n)
    for e in range(n_edges):
        for v in v_new:
            graph_rand_vertex = random.choice(max_deg_nodes)
            if not G_new.has_edge(v, graph_rand_vertex): then
                G_new.add_edge(v, graph_rand_vertex)
            end
    returnG_new (= G_out)
```

a vertex that will technically have a minimum degree, to design its inverse behavior a minimum degree vertex is chosen.

Algorithm 10: Remove a vertex with minimum degree from input graph

```
Input: Undirected graph G_in = (V_n, E_m)
Output: Undirected graph G_out = (V_n-1, E_m-removed_edges)
Function inv_add_node(input_graph: G_in):
    G_new = copy(G_in)
    min_deg_vertex = min(degree(G_new))
    neighbor_vertices = [G_new.neighbors(min_deg_vertex)]
    n = len(neighbor_vertices)
    removed_edges = [zip([min_deg_vertex]*n, neighbor_vertices)]
    G_new.remove_node(min_deg_vertex)
    return G_new, min_deg_vertex, removed_edges
```

Remove vertex with low degree centrality from given input graph:

A vertex with a low degree centrality is considered less important to a network [Pav+11]. Therefore, the operation is used to remove a vertex with the lowest degree centrality among all the vertices in a graph. In comparison to Algorithm 4,

which increases the centrality of a vertex, this Algorithm 11 removes the low-degree central vertex. A low-degree central vertex is considered during the deconstruction of a graph because centrality is a parameter of the vertex that grows with graph expansion from low to high. Hence each time the operation is applied for the deconstruction of a graph, the lowest degree central vertex is taken into consideration. Algorithm 11, takes graph $\mathcal{G}$ as the input parameter with n vertices and m edges, removes a low degree central vertex, and returns a new graph with $n-1$ vertices.

Algorithm 11: `Remove a vertex with lowest degree centrality from input graph`

Input: Undirected graph $\mathtt{G_{in}} = (\mathtt{V_n}, \mathtt{E_m})$
Output: Undirected graph $\mathtt{G_{out}} = (\mathtt{V_{n-1}}, \mathtt{E_{m-removed\_edges}})$
Function `inv_node_low_deg_cntra(`*input_graph: G_{in}*`):`

```
G_new = copy(G_in)
low_deg_vertex = min(degree_centrality(G_new))
G_new.remove_node(low_deg_vertex)
return G_new (= G_out)
```

Remove vertex with low closeness centrality from given input graph:

Opposite to high closeness centrality, vertices with low closeness centrality are less important to the network [Pav+11]. The vertex having the lowest closeness centrality among all the vertices in a graph is removed by the defined operation as shown in Algorithm 12. The vertex with low closeness centrality is selected to design an operation that is contrary to Algorithm 4, which connects a vertex to a high closeness central vertex. In the graph construction process, connecting a new vertex to a central vertex with a closeness property increases the weight of closeness from low to high. Hence to have inverse characteristics of Algorithm 4, that can deconstruct a graph, the low closeness centrality vertex is selected for removal. Algorithm 12, takes graph $\mathcal{G}$ as the input parameter with n vertices and m edges, removes the lowest closeness central vertex, and returns a new graph with $n-1$ vertices.

Remove vertex with low betweenness centrality from given input graph:

It removes a vertex from an existing graph that has the lowest betweenness centrality among all the vertices in the graph. Algorithm 13, is designed to deconstruct a graph and is opposite in nature to Algorithm 5, which is a graph construction operation. As

Algorithm 12: `Remove a vertex with lowest closeness centrality from input graph`

Input: Undirected graph $\mathtt{G_{in}} = (\mathtt{V_n}, \mathtt{E_m})$
Output: Undirected graph $\mathtt{G_{out}} = (\mathtt{V_{n-1}}, \mathtt{E_{m-removed\_edges}})$
Function `inv_node_low_close_cntra(`*input_graph: G_{in}*`)`:

```
G_new = copy(G_in)
low_close_cntra_vertex = min(closeness_centrality(G_new))
G_new.remove_node(low_close_cntra_vertex)
return G_new (= G_out)
```

discussed in previous algorithms, the characteristics of vertices, such as betweenness centrality increase from low to high as the complexity of the graph increases. While Algorithm 5, increases the betweenness centrality weight for a vertex, Algorithm 13 acts inversely and removes the low betweenness central vertex during deconstruction process. Algorithm 13, takes graph $\mathcal{G}$ as the input parameter with n vertices and m edges, removes the lowest betweenness central vertex, and returns a new graph with $n - 1$ vertices.

Algorithm 13: `Remove a vertex with lowest betweenness centrality from input graph`

Input: Undirected graph $\mathtt{G_{in}} = (\mathtt{V_n}, \mathtt{E_m})$
Output: Undirected graph $\mathtt{G_{out}} = (\mathtt{V_{n-1}}, \mathtt{E_{m-removed\_edges}})$
Function `inv_node_low_bwtn_cntra(`*input_graph: G_{in}*`)`:

```
G_new = copy(G_in)
low_bwtn_cntra_vertex = min(betweenness_centrality(G_new))
G_new.remove_node(low_bwtn_cntra_vertex)
return G_new (= G_out)
```

Remove vertex with low eccentricity centrality from given input graph:

A protein having low eccentricities has minimum action in the system [Pav+11]. Transferring this idea to a graph, the operation is used to remove a node from a graph with the lowest eccentricity value. Since Algorithm 6, is used for graph construction and increases the eccentricity of a vertex, the inverse would be to remove such vertices from the graph. The vertex is selected such that it has low eccentricity. Each time the operation is used for graph deconstruction, it chooses the vertex having minimum action in the system. Algorithm 14, takes graph $\mathcal{G}$ as

the input parameter with n vertices and m edges, removes the lowest eccentricity central vertex, and returns a new graph with $n-1$ vertices.

Algorithm 14: Remove a vertex with lowest eccentricity centrality from input graph

Input: Undirected graph $G_{in} = (V_n, E_m)$
Output: Undirected graph $G_{out} = (V_{n-1}, E_{m\text{-removed\_edges}})$
Function inv_node_low_ecc_cntra(*input_graph: G_{in}*):

```
G_new = copy(G_in)
low_ecc_cntra_vertex = min(eccentricity_centrality(G_new))
G_new.remove_node(low_ecc_cntra_vertex)
return G_new (= G_out)
```

Remove random vertex and edge from the given input graph:

An inverse operation to Algorithm 7, is to remove a random vertex and its connecting edges. Algorithm 15, depicts the removal of a random vertex and its connecting edges. The algorithm takes graph $\mathcal{G}$ as the input parameter with n vertices and m edges. It returns a new graph $\mathcal{G}$ with $n-1$ vertices.

Algorithm 15: Remove random vertex and edge from the given input graph

Data: Undirected graph $G_{in} = (V_n, E_m)$
Result: Undirected graph = G_{out}
Function Inv_add_one_node_random_edge(*input_graph:G_{in}*):

```
G_new= copy(G_in)
gcc = sorted(connected_components(G_new))
if len(gcc < 1) then
    raise Exception('Sorry, Input graph has unconnected subgraphs')
end
if len(G_new.nodes < 2) then
    raise Exception('Graph has number of nodes less than one')
end
node_list = get_node_list(G_new)
remove_node = random_vertex = random.choice(node_list)
neighbor_nodes = G_new.neighbors(removed_node)
removed_node_neighbors = [n for n in neighbor_nodes]
G_new.remove_node(random_vertex)
return G_new, removed_node, removed_node_neighbors
```

Remove one triangle and connecting edges from the given input graph:

Removing a triangle from an existing graph to achieve graph deconstruction with fewer steps is crucial, hence complex inverse operation is required. Algorithm 16, takes graph $\mathcal{G}$ and sorts all the vertices forming a triangle. One triangle which does not leave the output graph with unconnected components is removed along with edges connecting the triangle and the existing graph. The final output is a new graph $\mathcal{G}$ with $n-3$ vertices.

Algorithm 16: Remove one triangle and connecting edges

```
Data: Undirected graph G_in = (V_n, E_m)
Result: Undirected graph = G_out
Function inv_add_one_triangle_m_edges(input_graph:G_in):
    G_new = copy(G_in)
    if len(G_new.nodes) < 3 then
        raise Exception('Graph has number of nodes less than three')
    end
    removed_nodes = []
    removed_node_connections = []
    triangle_nodes = nx.triangles(G_new)
    tri_node = get_nodes_forming_triangle(triangle_nodes)
    if tri_node == 0 then
        raise Exception('No triangles in the graph found')
    end
    is_in_triangle = False
    have_triangle, triangle_nodes = has_triangle(G_new, tri_node)
    if have_triangle: then
        removed_nodes.extend(triangle_node)
        for node in triangle_node:
            neighbor_nodes = G_new.neighbors(node)
            removed_node_neighbors = [n for n in neighbors_nodes]
            nnl_size = len(removed_node_neighbors)
            nn = [node]*nnl_size
            removed_node_connections.extend(zip(nn,removed_node_neighbors))

            G_new.remove_node(node)
    end
    return G_new, removed_nodes, removed_node_connections
```

Remove k vertices and l edges from the given input graph:

Adding multiple vertices and edges in a single step during graph generation is more complex than working with a single vertex or edge. Similarly to deconstruct a graph removing k vertices and k edges can act as an inverse operation to Algorithm 9. This helps deconstruct complex structures in an existing graph with fewer steps. Algorithm 17, takes graph $\mathcal{G}$ with n vertices and m edges, removes k random vertices and their corresponding l edges. It returns a new graph with $n - k$ vertices.

Algorithm 17: Remove n nodes m edges

```
Data: Undirected graph G_in =(V_n, E_m)
Result: Undirected graph = G_out
Function inv_add_n_nodes_m_edges(input_graph:G_in , n_nodes=3):
    G_new = copy(G_in)
    if len(G_new.nodes < n_nodes) then
        raise Exception('Sorry, nodes less than one')
    end
    removed_nodes = []
    removed_node_connections = []
    for n in range(n_nodes):
        dgx = copy(G_in)
        degree_view = nx.degree(dgx)
        deg_view_dict = {key:value for key, value in degree_view}
        v_rand_choice = get_random_vertex(deg_view_dict.keys())
        dgx.remove_node(v_rand_choice)
        gcc = sorted(nx.connected_components(dgx))
        if len(gcc) == 1 then
            neighbors_nodes = G_new.neighbours(v_rand_choice)
            G_new.remove_node(v_rand_choice)
            removed_node_neigh = [n for n in neighbors_nodes]
            nn = len(removed_node_neigh)*[v_rand_choice]
            removed_node_connections.extend(zip(nn, removed_node_neigh))
             removed_nodes.extend([v_rand_choice])
        end
        if not len(gcc) == 1: then
            removed_node = get_min_key(deg_view_dict)
            neighbors_nodes = G_new.neighbours(removed_node)
            removed_node_neigh = [n for n in neighbors_nodes]
            nn = len(removed_node_neigh)*[v_rand_choice]
            removed_node_connections.extend(zip(nn, removed_node_neigh))
             removed_nodes.extend([removed_node])
            G_new.removed_node(removed_node)
        end
    return G_new, removed_nodes, removed_node_connections
```

3.3 Graph Construction Sequence Generation

As discussed before in section (3.2.1), decision operations are applied to an empty graph or an existing input graph to change the state of the current graph. In this research, graph operations are extended to nine different operations. When applied in sequence, these operations can construct a graph. Hence, a **construction sequence** is a sequence of decision operations that constructs a desired graph when taken into action. The actions of the construction sequence can also be seen as states of a finite-state machine (3.9).

A new method called the **graph deconstruction tree model** (3.3.3) and **graph deconstruction** (3.3.2) is envisioned, to obtain the construction sequences of graphs. The graph deconstruction tree model is an algorithm used to traverse the graph, and graph deconstruction is a process that deconstructs a graph using inverse operations while traversing the graph. This process gives a basis to generate construction sequences that represent any given input graph. The goal of generating a graph construction sequence is to provide a representation of a graph as a sequence of decisions that can be learned by neural networks.

3.3.1 Graph Construction & Sequences

The process of forming a graph progressively in a sequence of steps is graph construction. Various graph models such as Erdős–Rényi (ER), Watts-Strogatz (WS), and Barabási–Albert (BA) have pre-defined algorithms for graph construction. The deep generative models such as DGMG [Li+18], and DeepGG [SG21] generates graph by learning from exemplary graphs. In both approaches, the graph evolves with either one vertex or one edge in each step. To overcome this limitation, simple to complex decision operations (3.2.1) are designed that allow constructing a graph in fewer steps. The complex decision operations enhance the evolution of the complexity of a graph in a single step.

The graphical representation of graph evolution from an empty graph is shown in (Fig 3.1). Consider operations (add node) encoded as 0, (add edge) encoded as 1, (add n nodes m edges) encoded as 2, and (add triangle m edges) encoded as 3. As shown in the (Fig: 3.1), firstly a vertex (0) is added to a given empty graph $\mathcal{G}$. In the second step, a triangle with vertices $(1, 2, 3)$ and 3 edges $((0, 1), (0, 2), (0, 3))$ are connected. Finally a vertex (4) and 2 edges $((3, 4), (2, 4))$ are joined. This

illustration shows a complex operation can give rise to a complex graph in a single step of graph evolution.

Each step of graph progression is followed by a decision operation and a list of decision operations gives a final graph. Hence, this list of operations, if defined in an encoded sequence [**0 3 2**] is termed a ***graph construction sequence***. The decision operations of the construction sequence can be seen as states of the finite state machine, and are ***unparameterized sequence***. The edge connection between vertices is not incorporated in the sequence where the operation itself decides the evolution of edges, hence termed as *unparameterized sequence*. This allows the graph construction process to be more nondeterministic in nature. Figure (3.2 & 3.3) shows the same structure of the graph can be obtained through different paths, operations, and sizes of construction sequences.

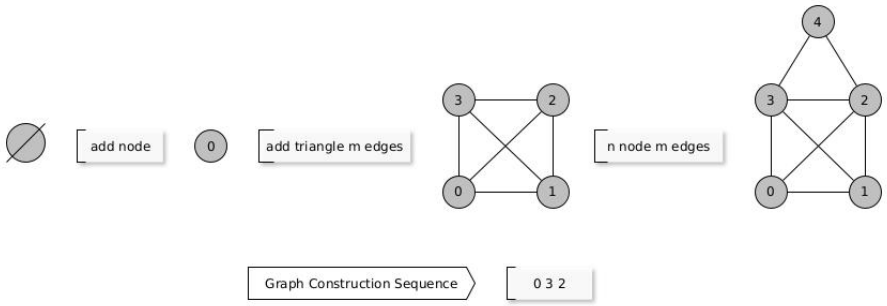

Figure 3.1: Shows construction of graph with decision operations. It starts with an empty graph and applies operations (add node, add n nodes m edges, and add triangle m edges) encoded as $(0, 2, 3)$ respectively. The figure depicts the construction of the desired graph in three steps using an unparameterized graph construction sequence [0 3 2].

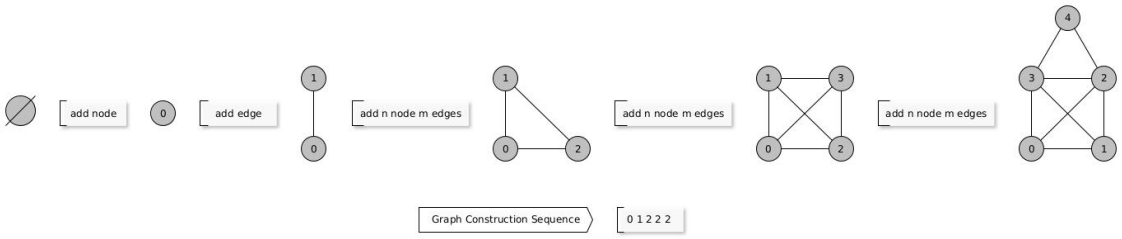

Figure 3.2: Shows construction of graph with decision operations. It starts with an empty graph and applies operations (add node, add edge, and add n nodes m edges) encoded as $(0, 1, 2)$ respectively. The figure illustrates the construction of the graph can be non-deterministic in nature. Though it follows a different path, gives the desired structure of the graph as in figure 3.3.

The deep auto-regressive model learns the construction sequence which represents

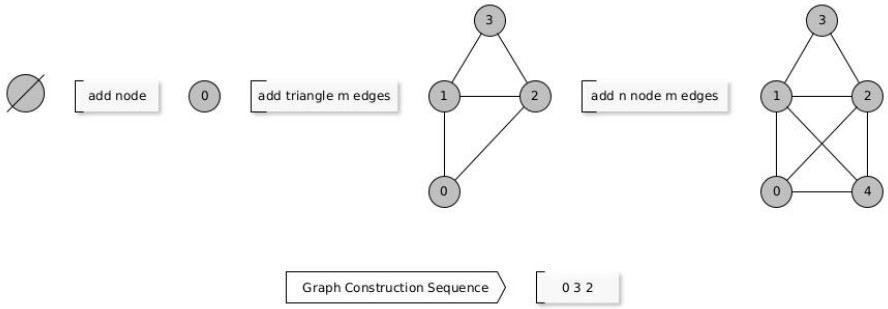

Figure 3.3: Shows construction of graph with decision operations. It starts with an empty graph and applies operations (add node, n nodes m edges, and add triangle m edges) encoded as $(0, 2, 3)$ respectively. The figure shows the construction of the graph can be non-deterministic in nature. Though it follows a different path, gives the desired structure of the graph as in figure 3.2. and in fewer steps.

the given graph and understands the graph structure and construction process. Hence to obtain the construction sequences for given input example graphs, a new technique called Graph Deconstruction 3.3.2 and Graph Deconstruction Tree Model 3.3.3 is used.

3.3.2 Graph Deconstruction

A process of removing a vertex, edge, or complex structure from a graph recursively until they are an empty graph is termed as ***Graph Deconstruction***. Eight different inverse decision operations (3.2.2) are modeled for graph deconstruction. Each step of deconstruction checks the validity of the inverse operation for the given input graph, as described in (3.3.3). As these inverse operations are used to deconstruct a graph until they are an empty graph, it results in a sequence of inverse operations titled ***deconstruction sequence***.

The purpose of the deconstruction sequence is to generate a construction sequence that follows a constructive way to recreate the deconstructed graph. Figure (3.4 & 3.5) illustrates that the same graph can be deconstructed in many possible paths and have distinct deconstruction sequences. Consider (inverse add node, inverse add m node m edges and inverse add triangle) are encoded as (0, 1, 2) respectively. The deconstruction sequence [1 2 0] for Figure (3.4) varies from [2 1 0] for Figure (3.5). Apart from the distinct nature, the length of the sequence can also vary due to complex inverse operations. The reversed form of the deconstruction sequence finally gives a construction sequence used for deep learning models and empirical

analysis. For the original input graph shown in figure (3.4, 3.5), the construction sequence would be [0 2 1] and [0 1 2] respectively that represents a sequence of decision operations that are reciprocal to itś corresponding inverse operations.

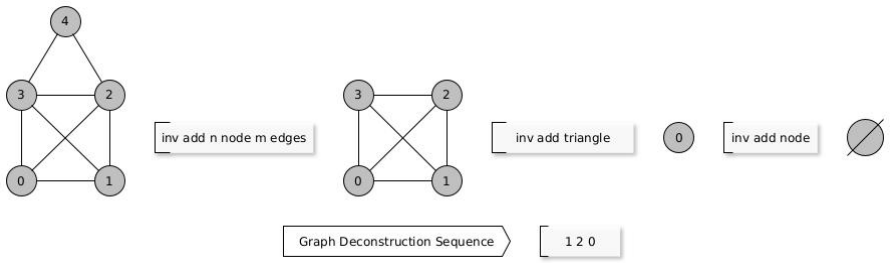

Figure 3.4: Shows deconstruction of the graph with inverse decision operations. It starts with an input graph and applies inverse operations (inverse add node, inverse add n nodes m edges, and inverse add triangle m edges) encoded as $(0, 1, 2)$ respectively. The figure shows the deconstruction of the graph with sequence [1 2 0].

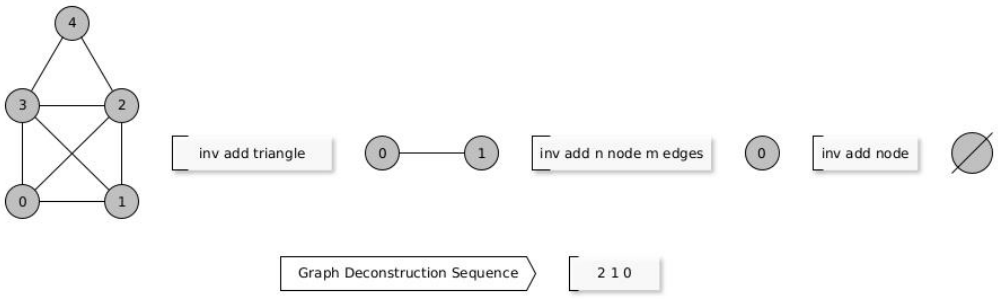

Figure 3.5: Shows deconstruction of the graph with inverse decision operations. It starts with an input graph and applies inverse operations (inverse add node, inverse add n nodes m edges, and inverse add triangle m edges) encoded as $(0, 1, 2)$ respectively. The figure shows the deconstruction of the graph with sequence [2 1 0].

3.3.3 Graph Deconstruction Tree Model

The previous discussion in section 3.3.2 and 3.3.1 provides a basic approach to graph deconstruction, construction, and the process of obtaining construction sequences. Since the given input graph in practice can have a large number of vertices and edges, a graph traversal strategy is required to obtain construction sequences. DeepGG [SG21] considers probabilistic model, breadth-first-search traversal, and depth-first-search traversal to obtain construction sequences of graphs. Whereas this research introduces a new method called ***graph deconstruction tree model***

(18) to obtain the sequences of graphs. Since the model is non-deterministic, it allows the exploration of the graph through many possible paths randomly. As a result of which the obtained construction sequences are not biased toward any specific traversal algorithm. Figure (3.6) shows a graphical representation of the working mechanism of the graph deconstruction tree model.

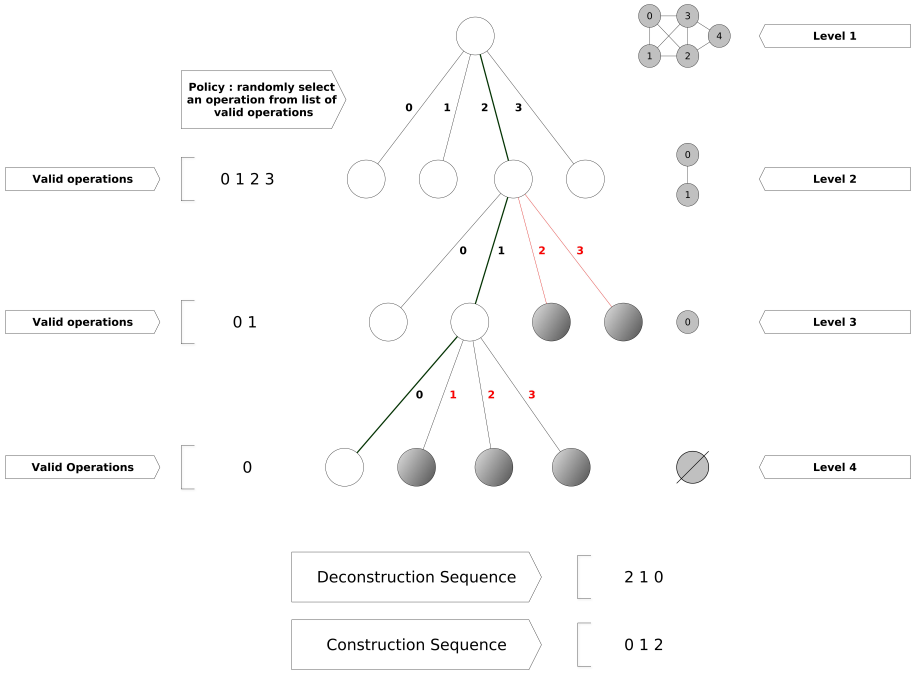

Figure 3.6: Graph deconstruction tree model(GDTM) deconstructs a graph through levels of iteration and valid operations. Generates a deconstruction sequence [2 1 0], when reversed results in construction sequence [0 1 2]. Now [0 1 2] represents a sequence of decision operations that are reciprocal to inverse operations.

The graph deconstruction tree model (GDTM) utilizes the concept of graph deconstruction (3.3.2) for the generation of the construction sequence. Consider inverse operations (inverse add node, inverse add one node random edges, inverse add n nodes m edges, inverse add triangle m edges) encoded as $(0, 1, 2, 3)$ respectively. As shown in figure (3.6), a graph $\mathcal{G}$ with 5 vertices and 8 edges is given as an input to GDTM. In the first iteration of GDTM as seen in Level 1 (3.6), all inverse operations are applied to the given input graph $\mathcal{G}$. Each operation outputs a new deconstructed graph. From the list of applied inverse operations, GDTM considers only those operations that output a deconstructed graph with no vertices left out of the graph connection. These operations are listed as *valid operations*. The first iteration gives valid operations as [0 1 2 3]. The non-deterministic nature of GDTM is governed by its policy of randomly selecting an operation from a list of valid operations. Hence, GDTM selects operation 2 and applies it to input graph $\mathcal{G}$ of which output is seen

in Level 2 (3.6).

The graph is fully deconstructed in three iterations resulting in a deconstruction sequence [2 1 0]. The reverse form of it [0 1 2], would be the construction sequence that represents the given graph $\mathcal{G}$. Now [0 1 2] represents a sequence of decision operations that are reciprocal to inverse operations. Since the inverse operations and GDTM are non-deterministic, the same graph $\mathcal{G}$ can be emptied in many possible paths. This allows a single graph to have several deconstruction and construction sequences. As a result, the deep learning model utilizing the sequences can capture many alternatives of graph construction for graph $\mathcal{G}$.

3.4 Learning to Classify Graph Decision Operations

In the research work of DGMG and DeepGG, the graph generation process is modeled in a state machine having three states namely *addnode*, *addedge*, *selectnode* for graph extension [Li+18] [SG21]. This research work extends the idea and creates more states as mentioned in (3.2). To understand how well an extended deep-state machine performs, graph neural networks are applied to learn the decision operations of the finite-state machine as a classification problem. Figure (3.7) is a graphical representation of the decision operations, where for each input graph an operation is applied to form an output graph. The output is modeled as per the characteristics of decision operations. This technique gives a base for creating graph datasets and learning the graph datasets as a problem of classification.

Following the principle of DeepGG and DGMG [SG21] [Li+18], 10 different local operations on graphs are designed as mentioned in section 3.2.1. These local operations are learned using the GNN classifier. Algorithm (20) shows the workflow to learn a classification model for local decision operations on graphs. The learning model architecture is designed using Graph convolutional networks (GCNs) (2.3.4). The final output of the GCNs architecture will be graph-level embedding denoted as $\mathbf{z}_{\mathcal{G}} \in \mathcal{R}^d$. The loss function is defined on $\mathbf{z}_{\mathcal{G}}$ embeddings, and stochastic gradient descent is applied to backpropagate the parameters of the loss function. The same decision operation when applied to a graph multiple times outputs diverse possible graphs as in Figure (3.8). Hence the graph data for training and testing are generated such that there are n possibilities of graph expansion or destruction. This allows the GNN model to capture as many alternatives of graph expansions and destruction. Finally, the learned model is used for the classification of local graph

Algorithm 18: Graph Deconstruction Tree Model

Data: Undirected graph $G_{in} = (V_n, E_m)$, inv_actions = [a1, a2,...,an]
Result: deconstruction_sequence = [a1, a3, a3, a1, a2, a6, a5,...]
Function graph_deconstruction_tree_model(*input_graph: G_{in}, inv_actions*):

```
G_new = copy(G_in)
n = len(G_new)
while n-1 do
    valid_random_actions = []
    while not valid_random_actions: do
        for act in range(len(inverse_actions)) :
            if inv_add_node == act and len(g.nodes)>= 1 then
                dgx,rm_node,rm_node_neighbors=inverse_actions[act](g)
                gcc = nx.connected_components(dgx)
                gcc = sorted(gcc, key=len, reverse=True)
                if len(gcc) == 1 then
                    valid_random_actions.append(act)
                end
            end
            if
             inv_add_one_node_rand_edge == act and len(g.nodes)>= 1
             then
                dgx,rm_node,rm_node_neighbors=inverse_actions[act](g)
                gcc = nx.connected_components(dgx)
                gcc = sorted(gcc, key=len, reverse=True)
                if len(gcc) == 1 then
                    valid_random_actions.append(act)
                end
            end
            triangle_nodes = nx.triangles(g)
            tri_node = [get_key_value(triangle_nodes)]
            if
             inv_add_triangle_n_edges == act and len(tri_node)>= 0
             then
                dgx,rm_node,rm_node_neighbors=inverse_actions[act](g)
                gcc = nx.connected_components(dgx)
                gcc = sorted(gcc, key=len, reverse=True)
                if len(gcc) == 1 then
                    valid_random_actions.append(act)
                end
            end
            if len(g.nodes) == 1 then
                dgx,rm_node,rm_node_neighbors=inverse_actions[act](g)
                valid_random_actions.append(act)
            end
    end
    deconstruction_sequence = graph_deconstruction_tree(inputs)
end
return deconstruction_sequence
```

Algorithm 19: Sub Algorithm for Graph Deconstruction Tree Model

Data: Undirected graph $G_{in} = (V_n, E_m)$, inv_actions = [a1, a2,...,an]
Result: deconstruction_sequence = [a1, a3, a3, a1, a2, a6, a5,...]
Function graph_deconstruction_tree(*input_graph: G_{in}, inv_actions, valid_random_actions*):

```
G_new = copy(G_in)
n = len(G_new)
valid_gcc = False
while not valid_gcc: do
    action = random.choice(valid_random_actions)
    if inv_add_node == act and len(g.nodes)>= 2 then
        node_num = len(g.nodes) - 1
        dgx, rm_node, rm_node_neighbors = inverse_actions[act](g)
        gcc = nx.connected_components(dgx)
        gcc = sorted(gcc, key=len, reverse=True)
        if len(gcc) == 1 then
            g = dgx.copy()
            valid_gcc = True
        end
    end
    if inv_add_one_node_rand_edge == act and len(g.nodes)>= 1
     then
        node_num = len(g.nodes) - 1
        dgx, rm_node, rm_node_neighbors = inverse_actions[act](g)
        gcc = nx.connected_components(dgx)
        gcc = sorted(gcc, key=len, reverse=True)
        if len(gcc) == 1 then
            g = dgx.copy()
            valid_gcc = True
        end
    end
    triangle_nodes = nx.triangles(g)
    tri_node = [get_key_value(triangle_nodes)]
    if inv_add_triangle_n_edges == act and len(tri_node)>= 0 then
        node_num = len(g.nodes) - 1
        dgx, rm_node, rm_node_neighbors = inverse_actions[act](g)
        gcc = nx.connected_components(dgx)
        gcc = sorted(gcc, key=len, reverse=True)
        if len(gcc) == 1 then
            g = dgx.copy()
            valid_gcc = True
        end
    end
end
deconstruction_sequence.append(action)
if len(g.nodes) == 1 then
    dgx, rm_node, rm_node_neighbors = inverse_actions[act](g)
    deconstruction_sequence.append(0)
end
return deconstruction_sequence
```

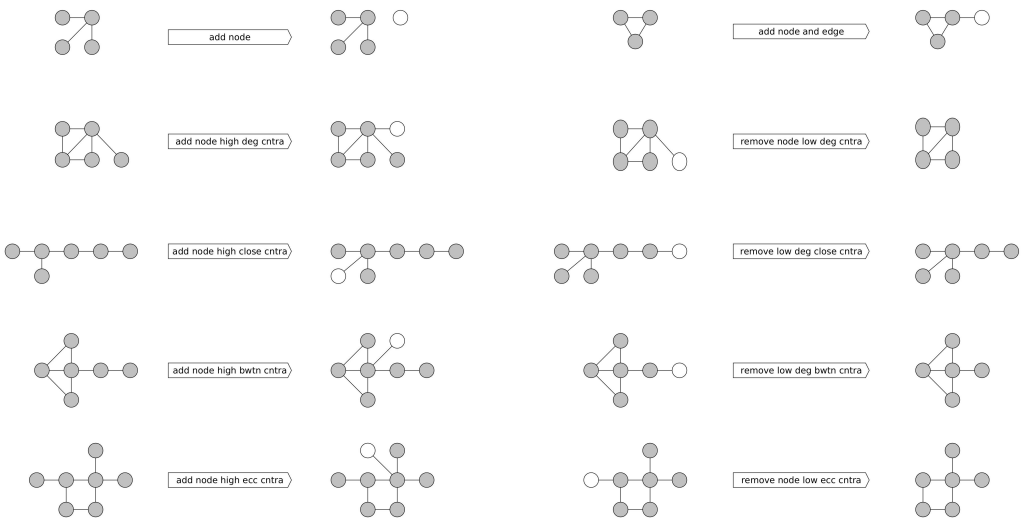

Figure 3.7: Graphical Representation of Decision Operations. Each operation is given an input graph which outputs a target graph. The input graph, target graph together with the decision operation are learned using the GNN classifier.

decision operations. This approach provides an understanding of how well an extended deep-state machine works which ultimately supports the design of generative models with a suitable number of states.

Algorithm 20: Algorithmic steps for learning local decision operations

- Generate n graphs G_i^m for each graph model where, $i = 1, ...n$, $m = 1, ..., 6$.
  - Graph models: ER, WS, BA
- For each possible decision operation, generate a set of target graphs corresponding to input graphs.
  - $G_i^{m,target} = DecisionOperation_j(G_i^m)$ where $j = 1, ..., 10$
  - For each operation j, $(G_i^m, G_i^{m,target})$ is obtained as dataset
- Create and learn a model that predicts the decision operation
  - $f(G_i^m) \longrightarrow G_i^{m,target}$

Evaluating how well the extended state machine works is essential to determine its effectiveness. A commonly used metric, accuracy (2.3.5) is considered to measure the classification model's performance.

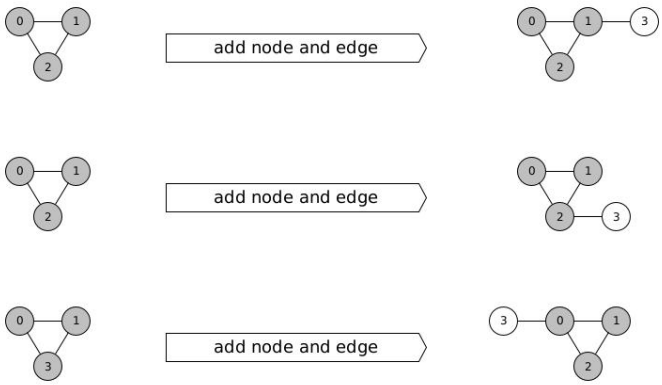

Figure 3.8: Graphical Representation of Decision Operations. Operation is given an input graph which outputs a target graph. The input graph, target graph together with the decision operation are learned using the GNN classifier. As shown in the figure, the same graph can be expanded through various possible paths, and graph data for training and testing are generated such that there is n number of possibilities of graph expansion. This allows the learning model to capture as many alternatives of graph expansion.

3.5 Deep State Machine (DSM)

The deep-state machine is an auto-regressive generative model of graph generation that inherits a strategy of learning decision operations from construction sequences with some major differences from DGMG ([Li+18]). The construction sequences are a sequence of descriptions of a given input graph. The deep-state machine in brief referred to as Deep Auto-Regressive Machines for Graph Construction (DarmGC) is designed to overcome existing limitations of DGMG ([Li+18]) and DeepGG ([SG21]). Here are list of key advancements done to further extend the previous research:

- A flexible finite-state machine in which more states can be incorporated. Five decision operations as states of the deep-state machine are taken into consideration for this research.
- The states of the deep-state machine can be loaded with a complex decision operation that allows manipulating the complex structure of a graph in a single state. This enforces the generation of graphs in fewer steps. For this research, three operations are designed to be complex.
- The construction sequences that are input for the model to learn the structure of graphs are unparameterized and obtained through a process called graph deconstruction tree model (GDTM) (3.3.3). It means the sequences only contain

sequences of decision operation and do not have the parameters that decide which nodes and edges are to be connected.

- Each state of the deep-state machine can decide its connectivity with the previous state output graph, during graph extension.

Figure (3.9) presents an extended state-machine that provides a base to design a deep-state machine based on a sequential generative model. It is termed extended because of its flexibility to incorporate multiple states. The deep-state machine learns the construction sequence using a sequential and auto-regressive process, during which a graph embedding is generated that replicates the graph formation process. As the deep-state machine transits, each state manifests a graph, and the embeddings of vertices and edges are combined to generate graph embedding using the message passing (2.3.2) framework. The output graph embedding of each transition is used to predict the next state. The figure shows four major steps including stop which denote the end of graph generation. Based on the current graph embedding, the deep-state machine decides the next state or decision operation.

Additionally, the deep-state machine is flexible enough to transition from any state to other existing states as it learns and generates unparameterized construction sequences. These sequences do not rely on parameters that are bound to give information regarding the connection between vertices and so does the deep-state machine. Therefore, the aim is to design a deep-state machine, that can be extended as needed and is not biased toward any traversal algorithm. Additionally, the extended state machine will be further utilized to generate datasets by randomly transitioning the states of the state machine (3.9). The process is referred to as **RTSM** . The resulting sequence of state transition will be treated as a graph construction sequence for evaluating the deep-state machine.

Encoding a graph in an embedding

Let $\mathcal{G}(\mathcal{V}, \mathcal{E})$ be a graph with vertices $v \in \mathcal{V}$ and edges $e \in \mathcal{E}$. The embedding for node v and edge e are represented as $\mathbf{h}_v \in \mathbb{R}^n$ and $\mathbf{h}_e \in \mathbb{R}^n$ respectively. Similarly, the graph embedding is defined as $\mathbf{h}_\mathcal{G} \in \mathbb{R}^k$. The hyper-parameters are set to $n = 16$ and $k = 2{\cdot}n$. Since the graph embedding $\mathbf{h}_\mathcal{G}$ embeds large information in comparison to node or edge embedding k must be comparatively larger than n. The deep-state machine starts with an empty graph and sets $\mathbf{h}_\mathcal{G} = (0, ..., 0)$. Multiple transitions

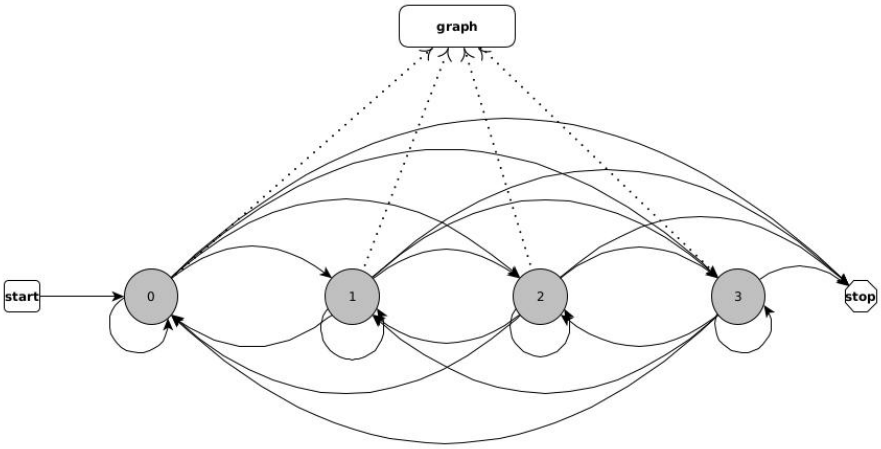

Figure 3.9: Shows extended state machine diagram. It provides a base to design a deep-state machine. Because of its flexibility to incorporate multiple states, it is termed extended.

of the states add structure to the graph and update the graph embedding $\mathbf{h}_{\mathcal{G}}$ in parallel.

Since the graph mutates progressively by adding new vertices v_{new} and edges e_{new}, DSM learns to initialize the vertex and edge using an initialization function based on Multi-Layer Perceptron (MLP) as in DeepGG ([SG21]). The initialization function is referred to as $f_{init,v}$ and $f_{init,e}$, respectively. Given a graph embedding vector $\mathbf{h}_{\mathcal{G}}$, a MLP with non-linear activation function σ, maps a graph embedding $\mathbf{h}_{\mathcal{G}} \in \mathbb{R}^{\mathcal{G}}$ to $\sigma(\mathcal{W}_{init,v} \cdot \boldsymbol{h}_{\mathcal{G}} + b_{init,v})$. It will be used further to decide whether to add a node or not. Similarly $\sigma(\mathcal{W}_{init,e} \cdot (\boldsymbol{h}_s \oplus \boldsymbol{h}_t) + b_{init,e})$ will be an deciding module to connect vertices. Where $\mathbf{h}_s$ and $\mathbf{h}_t$ represent source and destination vertices. Parameters such as $\mathcal{W}$ and b are to be learned as described in (2.2.1, 2.3.3).

Each time DSM transits to a new state, an embedding for the given state is initialized based on current graph embedding $\mathbf{h}_{\mathcal{G}}$. However, DSM has complex decision operations as a result of which multiple vertex and edge initialization is done in a single state of the deep-state machine. An example of updating $\mathbf{h}_{\mathcal{G}}$ with an embedding of a new triangle and no external edges from the triangle would be the consecutive initialization: $\mathcal{V}_{init}(f_{init,v0},\ f_{init,v1},\ f_{init,v2})$, and $\mathcal{E}_{init}(f_{init,e01},\ f_{init,e02},\ f_{init,e12})$. Each instance of $\mathcal{V}_{init}$ initializes the vertices of a triangle based on current graph embedding, whereas $\mathcal{E}_{init}$ initializes the edges between the vertices that form a triangle.

Learning deep-state machine

The deep-state machine initially starts with an initialization of an embedding $\mathbf{h}_{\mathcal{G}}$ for an unknown graph. Each iteration of training predicts the next state or step of a construction sequence. DSM initializes node representation $\mathbf{h}_v$ and edge representation $\mathbf{h}_e$ based on current graph embedding as per the given state or decision operation. These initialized vector representations create a mutation of the graph $\mathcal{G}_{previous}$ from the previous state. The current graph $\mathbf{G}_{current}$, is propagated based on a message-passing framework (2.3.2) to aggregate information from the local neighborhood and update the vertex representations. This process is known as graph propagation and is inherited from DGMG ([Li+18]). The approach of updating embeddings in DGMG is similar to GCN (2.3.4). The node embedding $\mathbf{h}_v$ receives a message from its neighboring node $\mathbf{h}_u$ as shown in equation (3.3):

$$\mathbf{m}_{v \to u} = \mathbf{W}_m \mathbf{concat}([\mathbf{h}_v, \mathbf{h}_u, \mathbf{x}_{v,u}]) + \mathbf{b}_m \tag{3.3}$$

Where, $\mathbf{x}_{v,u}$ is an embedding of the edge connecting v and u.

Since propagation is an iterative process, all vertices collect incoming messages and summarize (3.4) them to update their feature using Gated Recurrent Unit (3.5). Two rounds of graph propagation are performed for all vertex to update the vertex embeddings.

$$\mathbf{a}_u = \sum_{v:(v,u)\in\mathcal{E}} \mathbf{m}_{v \to u} \tag{3.4}$$

$$\mathbf{h}_u^{'} = \mathbf{GRU}(\mathbf{h}_u, \mathbf{a}_u) \tag{3.5}$$

Training and optimization of deep-state machine

The model decides state transitions based on categorical distributions, where the cross-entropy loss function is used to minimize the joint negative log-likelihood with stochastic gradient descent. The model is trained with a learning rate of $\eta = 0.0001$, for training epochs $\mathcal{T}_{epochs} = 3$, and 2 rounds of graph propagation. The maximum number of vertices to generate during graph generation is limited to $\mathcal{V}_{max} = 150$. During generative inference, a minimum value $\mathcal{V}_{min}$ can be set to stop the deep-state machine from transitioning to a stop state before reaching $\mathcal{V}_{min}$.

4 Experiments & Results

Learning the structure of graphs with neural models and applying those models for the link, node, and graph prediction or generation is a complicated task. For graph traversal, any vertices can be considered the starting point to reach the final vertex. This indicates a graph can be traversed in multiple paths. Therefore, learning multiple alternatives of graph traversal in a non-deterministic approach, such that the learned model is not biased toward any traversal algorithm becomes crucial. DGMG and DeepGG are some of the existing models that have shown significant progress in graph generation in an auto-regressive manner. However, as discussed in section (1.2), there are still open questions that are to be explored to overcome the limitations of existing research and discover new ways of learning graph structure and utilizing it for prediction and generation purposes. Hence, the following research is conducted to explore, understand and design new methods of graph generation.

4.1 Experiments

4.1.1 Learning to Classify Graph Decision Operations

Experimental Setup

To extend the state machine, nine decision operations (3.2.1) and eight inverse decision operations (3.2.2) have been designed. Out of the 17 different operations, five have been considered for the extension of the state machine to five states, as given in Table (4.1). Each state can also be viewed as a class of a classification task. To learn these states as a classification problem, different sets and combinations of graph datasets are generated by applying these graph manipulation operations.

The experiment uses three graph models, namely Erdos Renyi, Barabasi-Albert, and Watts-Strogatz. Table (4.2) displays Erdos-Reyni datasets, where one graph type

Type of Operation	Operation Name	Ref.
Decision Operation	add_node	1
Decision Operation	add_edge	2
Inverse Decision Operation	remove_node_low_deg_cntra	11
Decision Operation	add_node_high_deg_cntra	3
Inverse Decision Operation	remove_node_low_close_cntra	12

Table 4.1: States of the extended state machine (3.9). Each state is a graph manipulation operation and is considered the state of a state machine model. The operations are learned by graph neural network as a classification problem, where each state can also be considered a category in the classification task.

Erdos-Reyni Graph Datasets					
No. of Operations	**Dataset 1**	**Dataset 2**	**Dataset 3**	**Dataset 4**	**Dataset 5**
2	960	1280	1600	1920	2240
3	960	1200	1680	1920	2280
4	960	1280	1600	1920	2240
5	1000	1200	1600	2000	2200

Table 4.2: Shows graph datasets of Erdos-Reyni model. The first column is the number of operations applied to create the dataset. Each distinct graph in a dataset undergoes $n = 40$ manipulations with graph operations to generate the datasets. The remaining columns indicate the number of items in each dataset.

(ER) and 2, 3, 4, and 5 operations are applied respectively to generate data. Each distinct graph undergoes $n = 40$ manipulations with graph operations to generate n possible path of graph manipulation or representation. For example, 2 operations, 40 representations, and 1 graph type multiplied by 12 give 960 items as in Table (4.2). However, Table (4.3) contains a mixed dataset of Erdos-Reyni and Watts-Strogatz models. Hence the combination is formed with 2 graph types and 2, 3, 4, and 5 operations respectively. A similar pattern is followed in Table (4.4), where 3 graph types and 2, 3, 4, and 5 operations are used. As the graph datasets are ready, a classifier with two layers of Graph Convolution and a final linear layer is set up with the parameters specified in Table (4.5). The input parameters passed to the network are kept constant for all datasets, except for the number of classes. It varies depending on the number of operations used in each dataset.

Erdos-Reyni and Watts-Strogatz Mixed Graph Datasets					
No. of Operations	**Dataset 1**	**Dataset 2**	**Dataset 3**	**Dataset 4**	**Dataset 5**
2	960	1280	1600	1920	2240
3	960	1200	1680	1920	2240
4	960	1280	1600	1920	2240
5	800	1200	1600	2000	2400

Table 4.3: Shows mixed graph datasets of Erdos-Reyni & Watts-Strogatz model. The first column is the number of operations applied to create the dataset. Each distinct graph in a dataset undergoes $n = 40$ manipulations with graph operations to generate the datasets. The remaining columns indicate the number of items in each dataset.

Erdos-Reyni, Watts-Strogatz, & Barabasi-Albert Mixed Graph Datasets					
No. of Operations	**Dataset 1**	**Dataset 2**	**Dataset 3**	**Dataset 4**	**Dataset 5**
2	960	1200	1680	1920	2100
3	1080	1440	1800	2160	2520
4	960	1440	1920	2400	2880
5	1200	1800	2400	3000	3600

Table 4.4: Shows mixed graph datasets of Erdos-Reyni, Watts-Strogatz and Barabasi-Albert model. The first column is the number of operations applied to create the dataset. Each distinct graph in a dataset undergoes $n = 40$ manipulations with graph operations to generate the datasets. The remaining columns indicate the number of items in each dataset.

Parameters of GCN Architecture	
Parameter Type	**Value**
Batch Size	16
1^{st} Layer Input-Output Dimension	100-64
2^{nd} Layer Input-Output Dimension	64-32
Last Layer Dimension	32-No. of class
No. of Class	2, 3, 4, 5
Learning Rate	0.01

Table 4.5: Parameters used in GCN architecture for the training and testing of the classifier.

Accuracy of Model Trained and Test on Erdos-Reyni Datasets					
No. of Operations	**Dataset 1**	**Dataset 2**	**Dataset 3**	**Dataset 4**	**Dataset 5**
2	100	100	100	100	100
3	41.0	66.0	64.28	62.5	64.99
4	20.0	37.5	50.0	47.91	46.42
5	40.0	33.33	37.18	40.0	36.36

Table 4.6: Accuracy obtained from the learned model for the corresponding datasets and number of operations used in each dataset. The model is trained and tested on Erdos-Reyni graphs. The train and test ratio is set to (80:20)%.

Experimental Results

In this experiment, the classification model is trained on combinations of three graph types and five operations as mentioned in (4.1.1). The goal is to examine the performance of an extended-state machine with five operations. As the extended state machine problem is modeled as a classification task, Table (4.6) shows the evaluation metrics of the classification model.

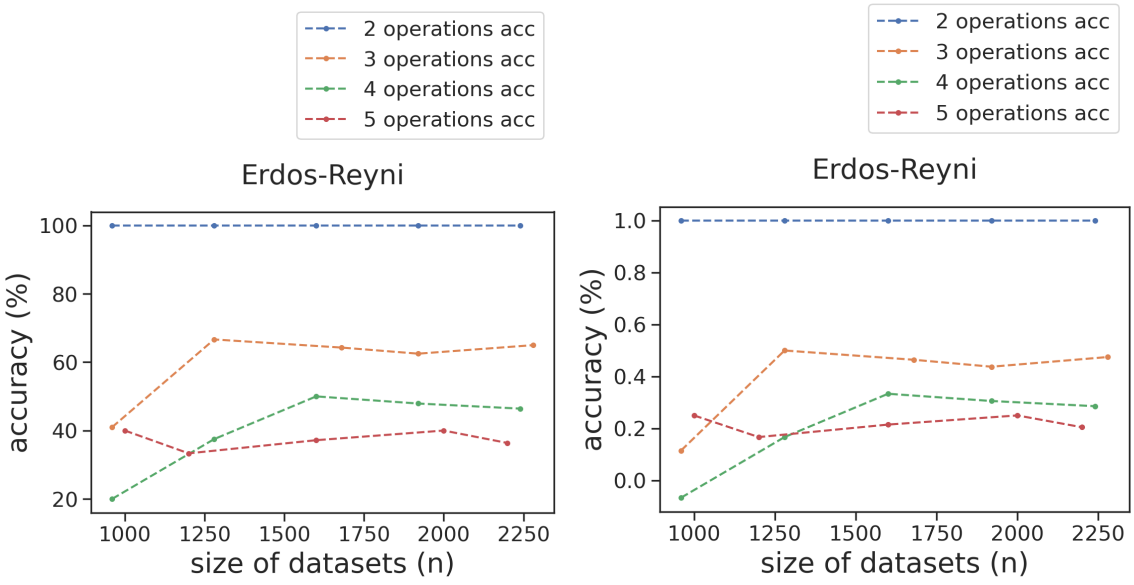

Figure 4.1: Shows the change in accuracy with a change in the size of datasets and the number of operations. The accuracy is obtained from a model trained and tested on 20 Erdos-Reyni Datasets (4.2) of various sizes.

Figure (4.1) shows accuracy obtained from the classification model trained and tested on (4.2) datasets. The x-axis of the plots represents the size of the datasets varied during the training and testing of the model, while the y-axis represents

achieved accuracy for corresponding graph datasets. The different colored lines connecting the points show the accuracy of the classification model with two, three, four, and five classes.

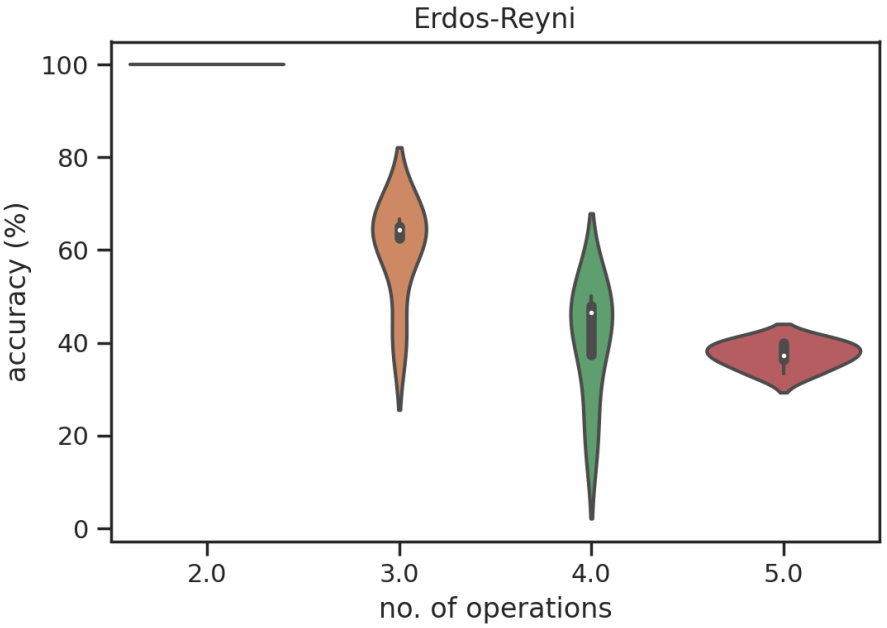

Figure 4.2: Shows the distribution of accuracy in relation to the number of operations obtained from the machine learning model. The model was trained and tested with graph datasets generated implying 2, 3, 4, and 5 operations (4.2).

The line chart in Figure (4.1), conveys that the increasing number of classes in a classification model leads to a decline in accuracy. Therefore this indicates the performance of the model to rightly classify is difficult with the number of classes to be discriminated against increases. On the other hand, varying the size of datasets does not show a significant impact on accuracy. To assure the results are not misleading, a normalized accuracy w.r.t. the number of operations is calculated as in Figure (4.1). Both of which support the same argument. The visualization in Figure (4.2) presents the density distribution of accuracy for classification models with different ranges of complexity. The symmetrical shape of the violin suggests that the accuracy follows a normal distribution for each number of operations. The density of accuracy for a model distinguishing two classes is higher in comparison to three, four, and five classes. It indicates models achieve a more accurate prediction for the two-class model. However, for models with three and four operations, there exist some outliers having significant differences from the median value. Though the accuracy for five operations is relatively low in comparison to three, and, four operations, the density of accuracy is higher at a point, suggesting the robustness

Accuracy of Model Trained and Tested on ER & WS Datasets					
No. of Operations	**Dataset 1**	**Dataset 2**	**Dataset 3**	**Dataset 4**	**Dataset 5**
2	100	100	100	100	100
3	58.33	66.66	59.52	62.5	66.66
4	41.66	37.10	50.0	47.91	46.42
5	0.0	33.33	25.0	30.0	33.33

Table 4.7: Accuracy obtained from the learned model for the corresponding datasets and number of operations used in each dataset. The model is trained and tested on combined sets of Erdos-Reyni & Watts-Strogatz Datasets (4.3). The train and test ratio is set to (80:20)%.

of the model.

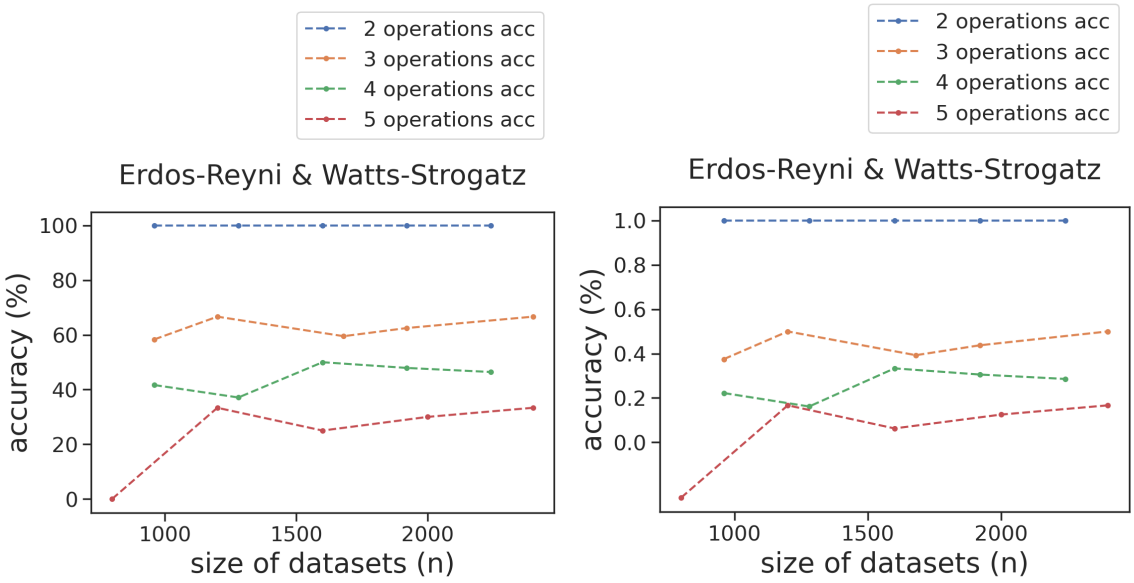

Figure 4.3: Shows the change in accuracy with a change in the size of datasets and the number of operations. The accuracy is obtained from a model trained and tested on 20 combined sets of Erdos-Reyni & Watts-Strogatz Datasets (4.3) of various sizes.

Table (4.7) and Figure (4.3) depicts a comparison of accuracy acquired through training and testing of the model across datasets (4.3). The plot demonstrates the accuracy remains constant for two operations, despite a varying number of datasets. However, as the number of operations increases, accuracy decreases for each setting i.e. three, four, and, five. It indicates the classification model is more effective to discriminate against a smaller number of operations, irrespective of the size of the datasets. In contrast, Figure (4.4) shows a symmetric violin plot, though the accuracy declines with an increase in the number of operations to be classified, the

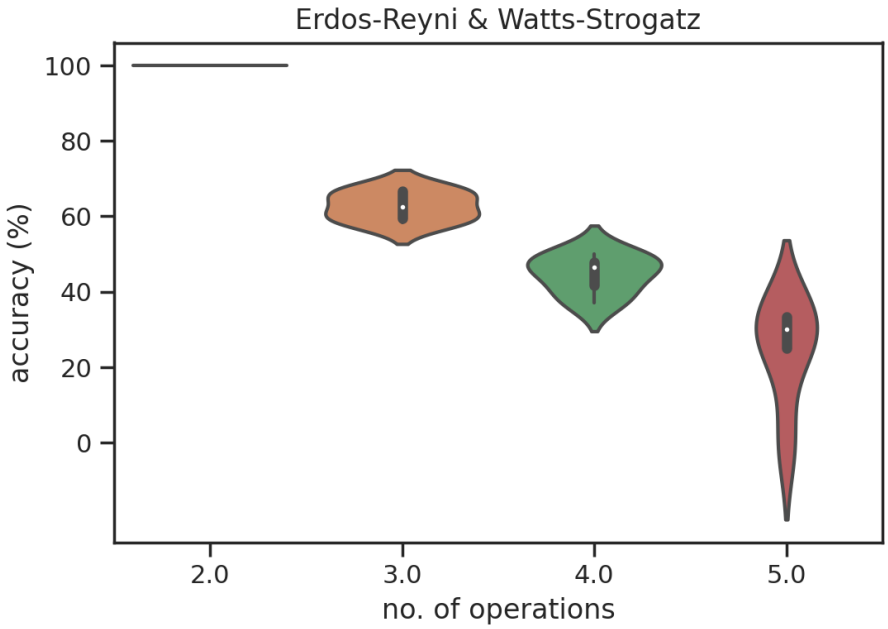

Figure 4.4: Shows the distribution of accuracy in relation to the number of operations obtained from the machine learning model. The model was trained and tested with graph datasets generated implying 2, 3, 4, and 5 operations (4.3).

distribution of accuracy is centered around the median value, suggesting the robustness of classifier. Table (4.7) presents the accuracy of Dataset 1 with five operations, indicating an accuracy of 0.0. It is discovered that the size of the dataset producing this result is relatively very small. Both plots (4.3, 4.4), suggest maintaining the minimum size of graph datasets is crucial in order for the machine learning model to learn the underlying patterns.

Consistent with previous experiments, Table (4.8) and Figure (4.5) depicts accuracy derived from training and testing of the (4.4) Datasets. The results indicate the classification model performs precisely when the number of operations being classified is relatively low, as seen in (4.5). Similar to the prior experiments, the result illustrates that the size of the dataset does not significantly affect the accuracy for each classification setting. The symmetric violin plot (4.6) demonstrates a high density of accuracy centered around the median and relatively fewer outliers than in previous experiments. This suggests that the model is more robust with an increase in types of graphs, irrespective of obtained accuracy.

Here is a list of results and discoveries that summarize the experiment:

1. The classification model is more effective in discriminating against a fewer

Accuracy of Model Trained and Tested on ER, WS & BA Datasets					
No. of Operations	**Dataset 1**	**Dataset 2**	**Dataset 3**	**Dataset 4**	**Dataset 5**
2	99.47	100	97.02	89.32	100
3	62.96	69.44	66.66	59.95	69.84
4	41.66	44.44	41.6	50.0	50.34
5	33.33	33.33	33.25	41.0	33.33

Table 4.8: Accuracy obtained from the learned model for the corresponding datasets and number of operations used in each dataset. The model is trained and tested on combined sets of Erdos-Reyni, Watts-Strogatz & Barabasi-Albert Datasets (4.4). The train and test ratio is set to (80:20)%.

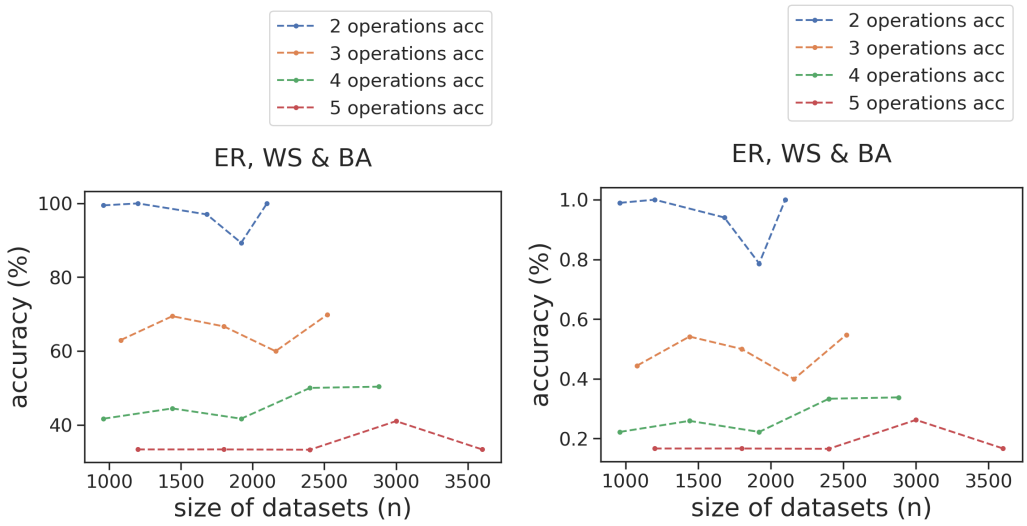

Figure 4.5: Shows the change in accuracy with a change in the size of datasets and the number of operations. The accuracy is obtained from a model trained and tested on 20 combined sets of Erdos-Reyni, Watts-Strogatz & Barabasi-Albert Datasets (4.4) of various sizes.

number of operations. As the number of operations increases, there is a decline in accuracy.

2. The dataset size does not significantly impact accuracy unless the minimum amount of dataset for the model to learn the patterns is met.

3. The predictions of the classification model get stable as the number of graph types is increased in the datasets, regardless of the obtained accuracy. This suggests that the model is more robust with an increase in the types of graphs.

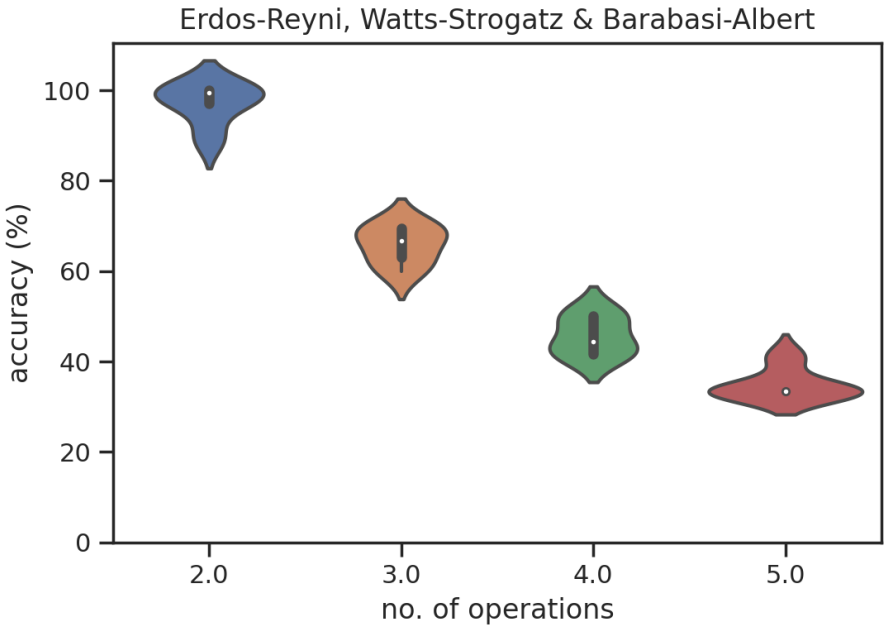

Figure 4.6: Shows the distribution of accuracy in relation to the number of operations obtained from the machine learning model. The model was trained and tested with graph datasets generated implying 2, 3, 4, and 5 operations (4.4).

4.1.2 Learning Construction Sequences with Extended Deep State Machine

Experimental Setup

The state machine-based deep generative models (DeepGG [SG21]) for graph generation consisted of three states, namely *addnode*, *addedge*, and *selectnode* to expand the graph. To extend the idea, this experiment establishes a deep-state machine with five states as in Table(4.9), adding complexity to some states. Each state is referred to as a graph decision operation and is designed to add complexity. The experiment considers three graph models, Erdos Reyni, Barabasi-Albert, and Watts-Strogatz, to train the extended deep-state machine. The dataset size is $n = 1000$ for each graph model (4.10), and the deep-state machine is evaluated on 500 samples of each graph model, including graphs generated with random transitioning of extended state-machine (RTSM) (3.5). However, the deep-state machine actually learns the representations of graphs in the form of construction sequences, hence each distinct graph in a dataset undergoes $i = 5$ iterations of GDTM (3.3.3) to generate the construction sequence. The deep-state machine is trained with a set of $n * i = 5000$ construction sequences, as of result of which it learns $i = 5$ different

Type of Operation	Operation Name	Ref.
Decision Operation	add_node	1
Decision Operation	add_one_node_rand_edge	7
Decision Operation	add_triangle_n_edges	8
Decision Operation	add_n_nodes_m_edges	9
Decision Operation	stop	–

Table 4.9: Shows complex states of the extended deep state machine. Each state is a graph manipulation operation and is considered the state of the deep-state machine. The operations are instances of construction sequences learned by the deep-state machine for the graph generation.

Datasets for experiment 2				
Graph Model	**Training**	**Evaluation**	**No. of Nodes**	**Parameters**
Erdos-Reyni	1000	500	83	(p=0.1)
Watts-Strogatz	1000	500	83	(p=0.7, k=7)
Barabasi-Albert	1000	500	83	–
RTSM (3.5)	–	500	83 (Max.)	–

Table 4.10: Shows ER, WS, and BA graphs for training and evaluation of extended deep state machine. RTSM datasets are only used for evaluation. RTSM: The datasets are created with random transitioning of the extended state machine (3.9, 3.5).

alternative paths of graph construction for each distinct graph. Degree distribution and probability distribution of graphs including Maximum Mean Discrepancy (MMD) (2.3.5) are used for the evaluation of the deep-state machine.

Experiment Results

The Figure (4.7, 4.8), depicts the degree distribution and probability density of graphs generated using the deep-state machine, evaluation datasets from Erdos-Reyni, and random transitioning of extended state-machine (RTSM) (3.5). The degree distribution of the Erdos-Reyni appears symmetrically inclined when $p = 0.1$ approaches 1. With deep-state machine-generated graphs trained on $n = 5000$ sequences of Erdos-Reyni graphs, the degree distribution does follow a power law but differs from ground truth (i.e: Erdos-Reyni). Also, the RTSM-generated graphs follow a power law. This demonstrates that the deep-state machine can learn the construction sequence based on the GDTM (3.3.3) model and generate graphs according to the learned sequences. Yet, the degree distribution of DSM-generated

graphs differs from Erdos-Reyni graphs. It indicates graph deconstruction and construction approaches with GDTM seem to fail in capturing the properties of the training graphs. The decision operation during graph expansion is decided by a deep-state machine, but the connectivity is defined by the graph decision operation itself. Therefore, it can be a reason the distribution varies. The probability density of graphs (4.8) also supports the results shown in the degree distribution.

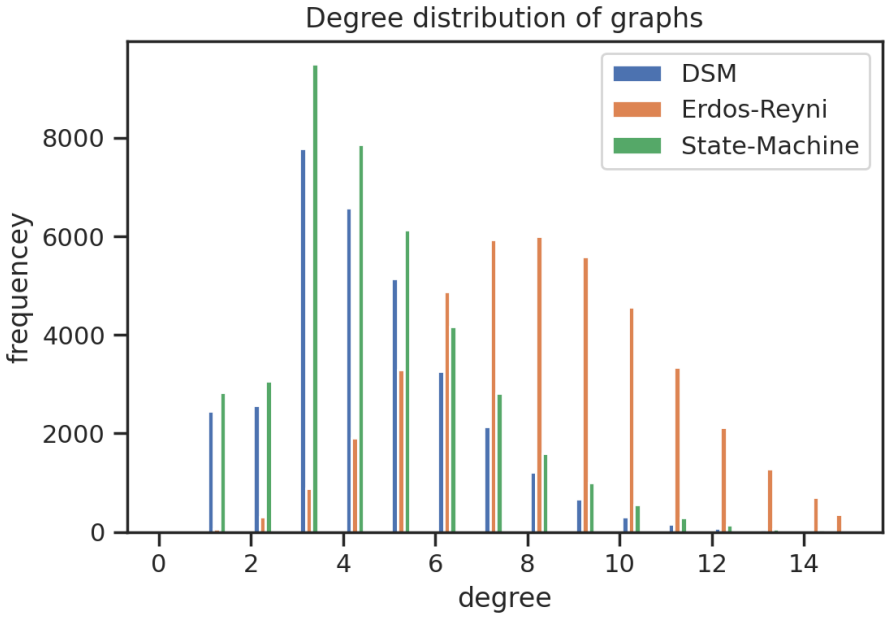

Figure 4.7: Shows degree distribution of graphs generated with the deep-state machine (DSM), Erdos-Reyni, and State-Machine (RTSM) model.

Table (4.11) illustrates the statistical similarity of graphs generated by deep-state machine-generated and RTSM (3.5), with an MMD of 0.075. This value is significantly lower than the MMD of 0.37 obtained compared to DSM and Erdos-Reyni graphs. Although the MMD value of 0.37 is higher than 0.075, it is still significantly less than 1 which is the highest possible value of MMD. This suggests that the deep-state machine learns the graph construction sequences but fails to capture the properties of trained graphs. Despite some level of dependency on decision operation for the graph generation, this demonstrates that an extended deep-state machine effectively generates graphs but with some level of dissimilarity to the target graphs. The deep-state machine trained with Watts-Strogatz graphs and its corresponding evaluation datasets, as shown in Figure (4.9, 4.10) and Table (4.12), exhibits similar pattern as in the previous experiment. Additionally, the deep-state machine trained with Barabasi-Albert graphs compared to evaluation datasets shows it follows power-law degree distribution as illustrated in Figure (4.11). However, the MMD value of 0.36 between DSM-generated graphs and Barabasi-Albert graphs

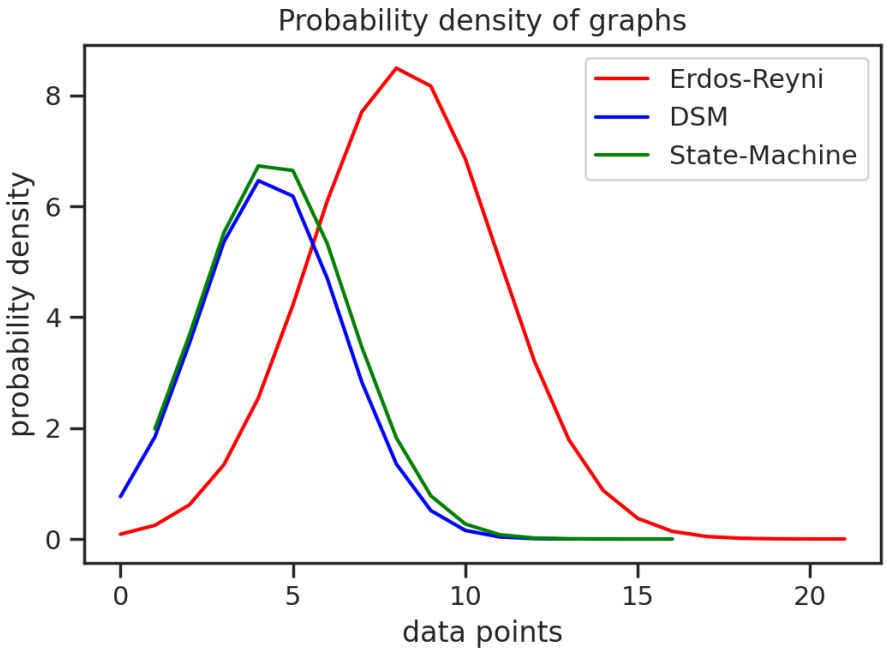

Figure 4.8: Shows probability density of graphs generated with the deep-state machine, Erdos-Reyni, and State-Machine (RTSM) (3.5) model.

Maximum Mean Discrepancy		
	ER	**RTSM**
DSM Generated Graphs	0.37	0.075

Table 4.11: Shows MMD between deep-state machine (DSM) generated graphs versus Erdos-Reyni and RTSM (3.5). The deep-state machine is trained with a set of $n = 5000$ graph construction sequences obtained from the Erdos-Reyni graph.

suggests some differences between ground truth and generated graphs supporting the facts of previous experiments.

Here is a list of results and discoveries that summarize the experiment:

- A deep-state machine learns the graph representations in the form of graph construction sequences, where each graph contains $n = 83$ vertices. The deep-state machine is capable to generate graphs with a maximum of $n = 83$ vertices.

- A deep-state machine is trained with construction sequences of Erdos-Reyni, Barabasi-Albert, and Watts-Strogatz graphs. However, the graph generated by the deep-state machine exhibits similarity in a number of vertices but dissimilarity in-degree distribution.

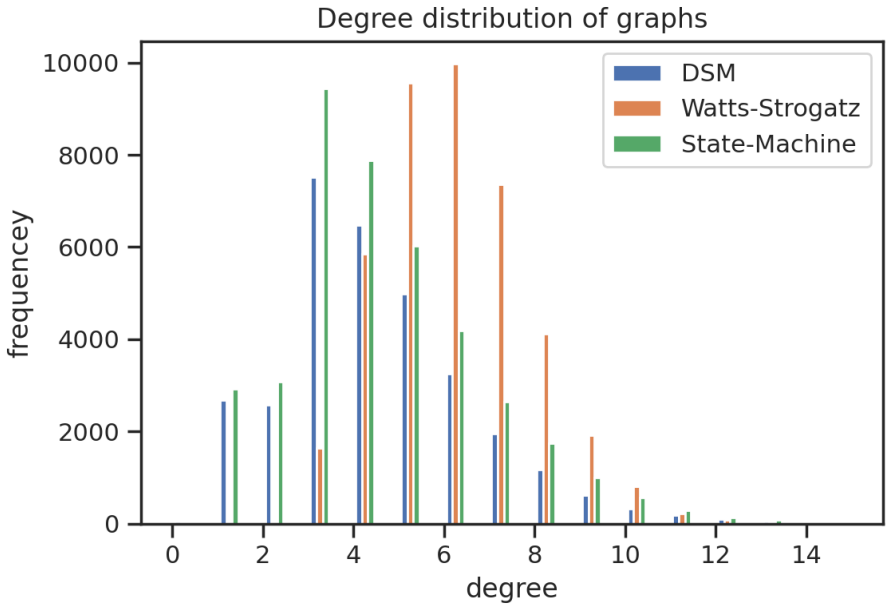

Figure 4.9: Shows degree distribution of graphs generated with the deep-state machine (DSM), Watts-Strogatz, and State-Machine (RTSM model) (3.5).

Maximum Mean Discrepancy		
	WS	**RTSM**
DSM Generated Graphs	0.29	0.093

Table 4.12: Shows MMD between deep-state machine (DSM) generated graphs versus Watts-Strogatz and State-Machine (RTSM) (3.5). The deep-state machine is trained with a set of $n = 5000$ graph construction sequences obtained from Watts-Strogatz graphs.

- The degree distribution of deep-state machine-generated graphs always follows the scale-free property of the graph.
- The maximum mean discrepancy between the deep-state machine model generated graph and evaluation graph indicates some degree of structural dissimilarity. This implies that the deep-state machine is unable to generate structurally similar graphs.

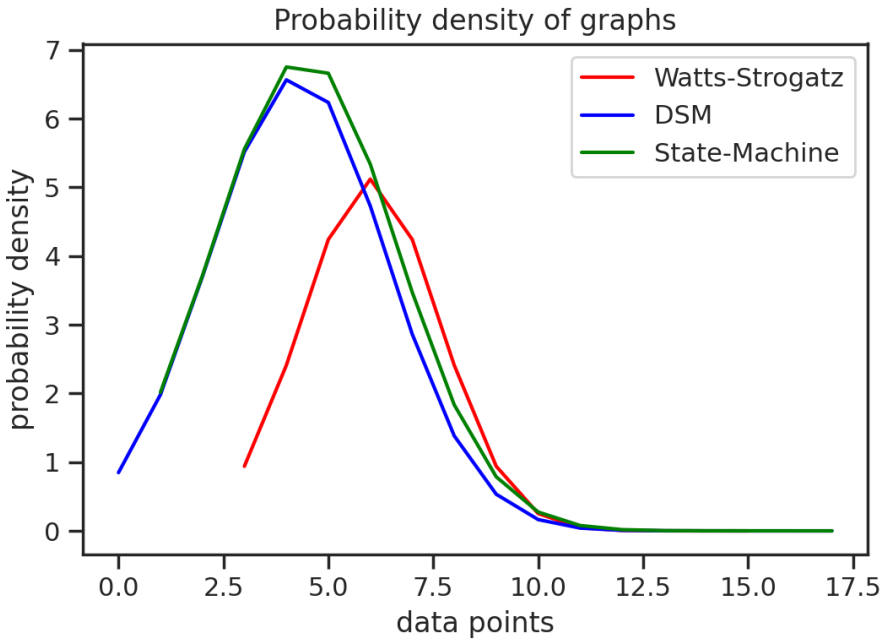

Figure 4.10: Shows probability distribution of graphs generated with the deep-state machine (DSM), Watts-Strogatz, and State-Machine (RTSM) (3.5).

Maximum Mean Discrepancy		
	BA	**RTSM**
DSM Generated Graphs	0.36	0.12

Table 4.13: Shows MMD between deep-state machine (DSM) generated graphs versus Barabasi-Albert and State-Machine (RTSM) (3.5). The deep-state machine is trained with a set of $n = 5000$ graph construction sequences obtained from Barabasi-Albert graphs.

4.1.3 Empirical Evaluation of Decision Operations and Graph Construction Sequences.

Experimental Setup

The previous experiments mostly deal with learning decision operations, graph construction sequences, and deep-state machines for graph generations. Hence understanding the patterns of decision operations is crucial to get insights into the previous experiments. For this purpose, all datasets created for the experiment (4.1.2) are taken into consideration including the data generated by the deep-state machine. Additionally, a new set of datasets for Erdos-Reyni, Watts-Strogatz, and Barabasi-Albert models is created each having 1000 graphs. Where each set of graphs is

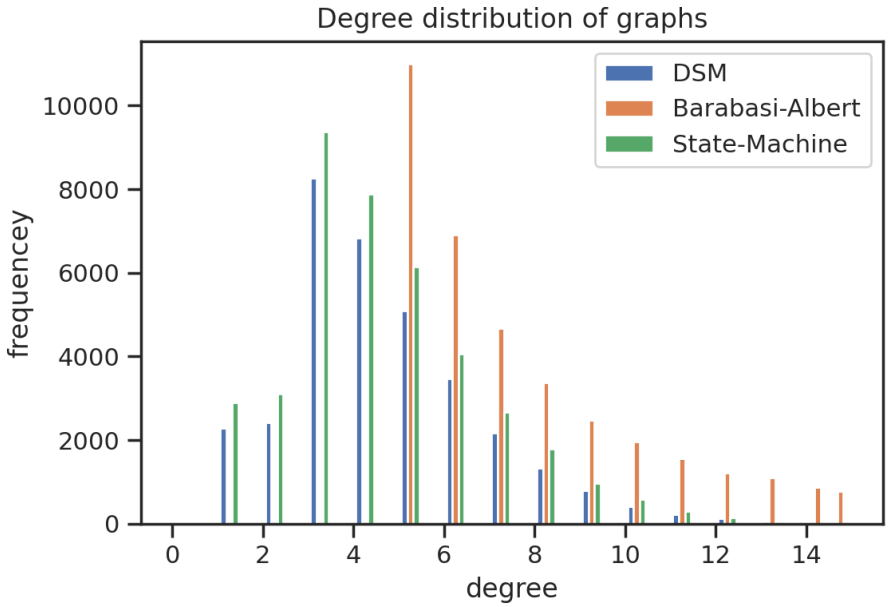

Figure 4.11: Shows degree distribution of graphs generated with the deep-state machine (DSM), Barabasi-Albert, and State-Machine (RTSM) (3.5).

converted to $n = 5000$ graph construction sequences. Distributions of decision operations in the construction sequences are visualized to get insights.

Experimental Results

Figure (4.13, 4.14, 4.15) depicts the count of each operation in each set of graph construction sequence of dataset A and B. The graph construction sequences are obtained through the GDTM model. The figures suggest the consistency of the GDTM model in generating graph construction sequences. Additionally (4.16, 4.17, 4.18) demonstrates the distribution of operation as per sequence location for $n = 5000$ construction sequences. Each plot gives the idea that a graph construction always starts with a node and gradually the distribution of operations becomes diverse in each location. It also shows the consistency of the GDTM model to generate a sequence with similar distribution for various graph models and a sequence length very close to each other.

The deep-state machine learns the graph construction sequences and its distribution of operation as shown in (4.16, 4.17, 4.18), and generates graph construction sequences with the distribution of operations illustrated in Figure (4.19, 4.20, 4.21). In a comparison of operation distribution between training data and DSM-generated

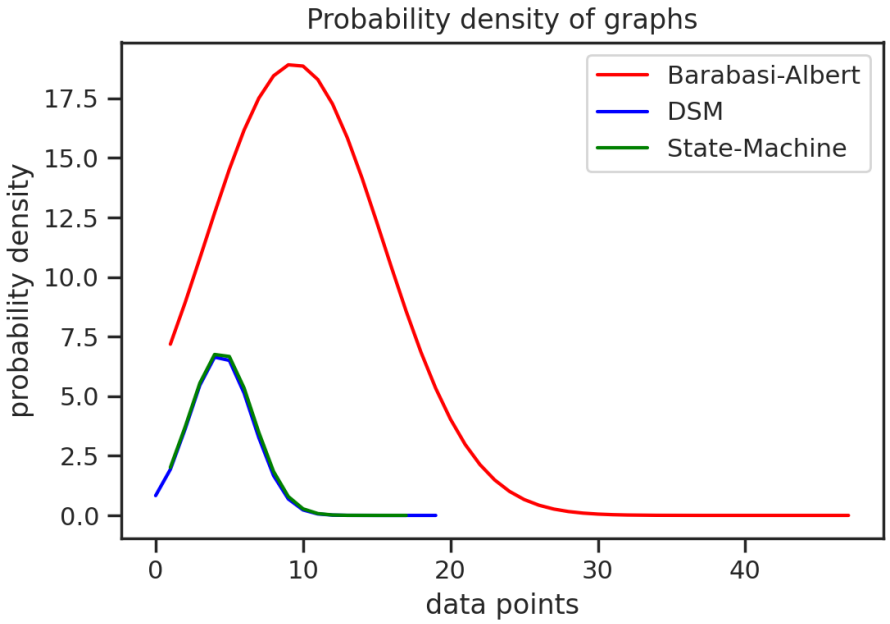

Figure 4.12: Shows probability distribution of graphs generated with the deep-state machine (DSM), Barabasi-Albert, and State-Machine (RTSM) (3.5).

data, there are some visual similarities in the distribution and DSM is able to decrease the sequence length yet generate a graph with $n = 83$ vertices.

Lastly, Figure (4.22, 4.23, 4.24) exhibits the operation distribution of graph construction sequences generated with random transitioning of state-machine (RTSM)(3.5). The visual representation shows, each operation is near to equally distributed from the very beginning of the sequence. This distribution significantly varies from the operation distribution of training data and DSM-generated data. Hence, the deep-state machine is able to learn the distribution of operations from trained data obtained with the GDTM model and generate sequences with a similar distribution. At the same time, it can be said that the DSM does not generate a construction sequence randomly, but generates based on its training.

Here is a list of results and discoveries that summarize the experiment:

- GDTM model for graph construction sequences is robust in its policy. The operation count and the operation distribution shows consistency in multiple datasets.
- The deep-state machine is able to learn and generate graph construction sequences with similar operation distribution.

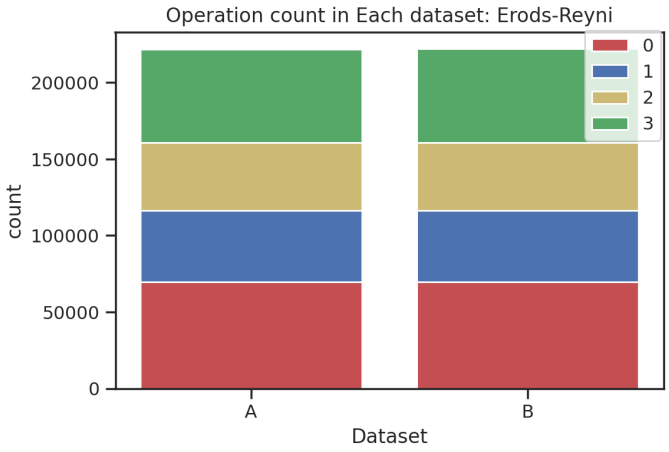

Figure 4.13: Shows operation count for Erdos-Reyni graph datasets: A and B. Each category of operation in the graph construction sequences obtained from A and B is counted and visualized.

- DSM generated graph construction sequences length is relatively less than ground truth, yet generates a graph with $n = 83$ vertices.

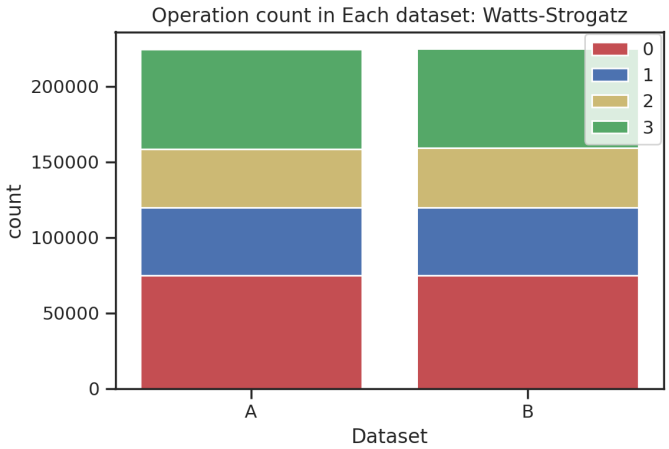

Figure 4.14: Shows operation count for Watts-Strogatz graph datasets: A and B. Each category of operation in the graph construction sequences obtained from A and B is counted and visualized.

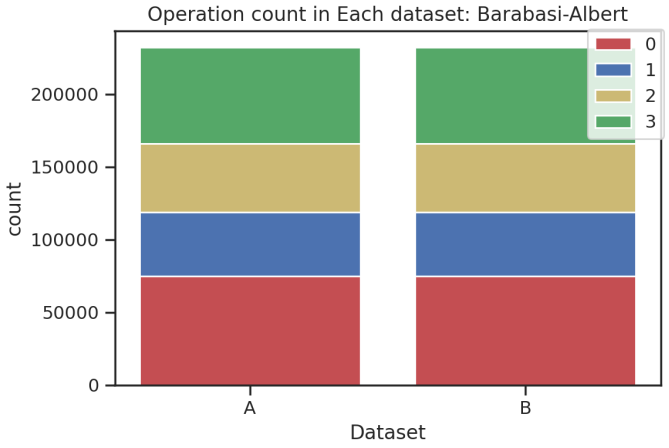

Figure 4.15: Shows operation count for Barabasi-Albert graph datasets: A and B. Each category of operation in the graph construction sequences obtained from A and B is counted and visualized.

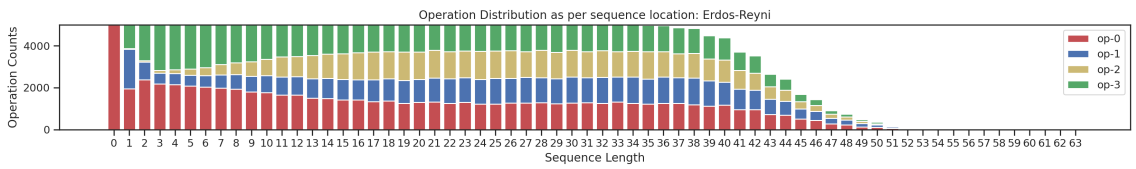

Figure 4.16: Shows the distribution of operation for Erdos-Reyni graph construction sequences as per sequence location. It depicts distribution for $n = 5000$ construction sequence of variable length.

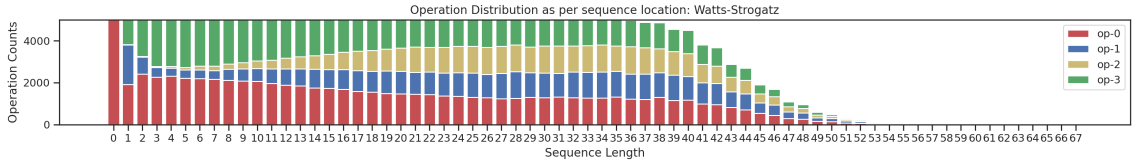

Figure 4.17: Shows the distribution of operation for Watts-Strogatz graph construction sequences as per sequence location. It depicts distribution for $n = 5000$ construction sequence of variable length.

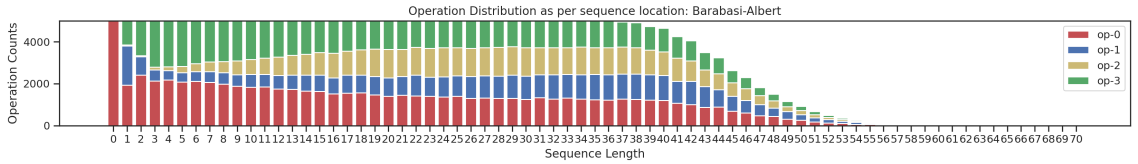

Figure 4.18: Shows the distribution of operation for Barabasi-Albert graph construction sequences as per sequence location. It depicts distribution for $n = 5000$ construction sequence of variable length.

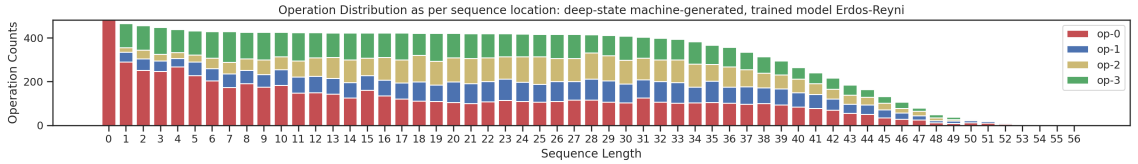

Figure 4.19: Shows the distribution of operation for deep-state machine-generated construction sequences as per sequence location. The deep-state machine is trained with Erdos-Reyni. It shows the distribution for $n = 500$ construction sequence of variable length.

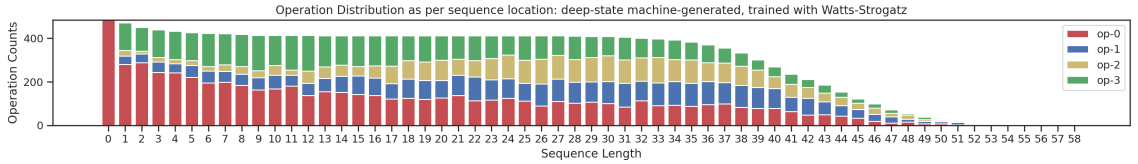

Figure 4.20: Shows the distribution of operation for deep-state machine-generated construction sequences as per sequence location. The deep-state machine is trained with Watts-Strogatz. It shows the distribution for $n = 500$ construction sequence of variable length.

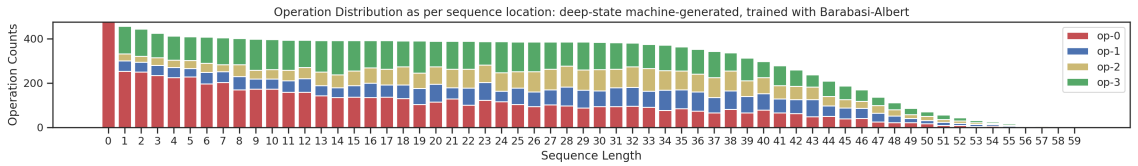

Figure 4.21: Shows the distribution of operation for deep-state machine-generated construction sequences as per sequence location. The deep-state machine is trained with Barabasi-Albert. It shows the distribution for $n = 500$ construction sequence of variable length.

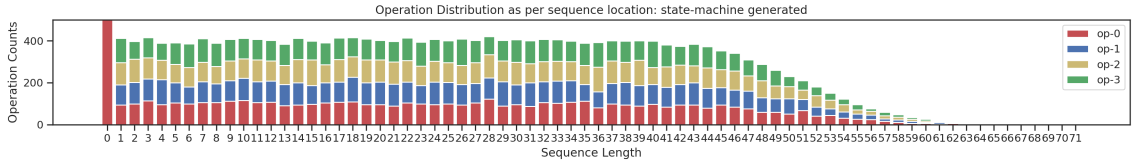

Figure 4.22: Shows the distribution of operation for state-machine (RTSM) (3.5) generated construction sequence through random transitioning. It depicts the distribution for $n = 500$ construction sequence of variable length.

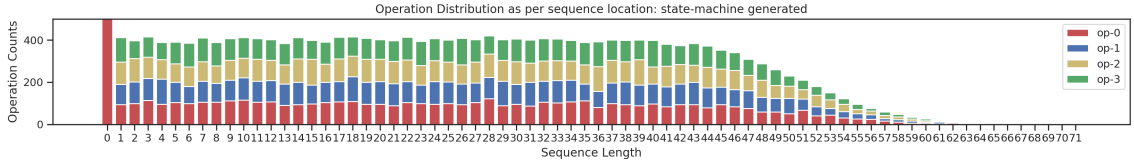

Figure 4.23: Shows the distribution of operation for state-machine (RTSM) (3.5) generated construction sequence through random transitioning. It depicts the distribution for $n = 500$ construction sequence of variable length.

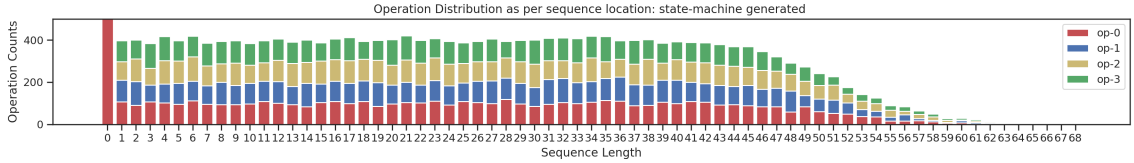

Figure 4.24: Shows the distribution of operation for state-machine (RTSM) (3.5) generated construction sequence through random transitioning. It depicts the distribution for $n = 500$ construction sequence of variable length.

5 Discussion

The first experiment (4.1.1) demonstrates that the classification model performs better at discriminating against a fewer number of operations with a reasonable dataset size. These findings when transferred to a deep-state machine, suggest that it can be extended and generalized, but is advisable to have a minimum number of states possible. However, the effectiveness of generalization and extension depends on how different are the states of the deep-state machine. In this experiment, each state is applied to a graph $\mathcal{G}_{in}$ and outputs $\mathcal{G}^1_{out}, \mathcal{G}^2_{out}, \mathcal{G}^3_{out}$ and so on. Since the edit distance between $(\mathcal{G}_{in}, \mathcal{G}^1_{out})$, $(\mathcal{G}_{in}, \mathcal{G}^2_{out})$, and $(\mathcal{G}_{in}, \mathcal{G}^2_{out})$ is very close to each other, differentiating which state is applied for the graph manipulation becomes challenging. As a result, extending deep-state machines with states that are similar to each other may not yield a well-performing deep-state machine. Therefore, it is recommended to perform a similar experiment as (4.1.1), to ascertain the appropriate states for a deep-state machine. That being the case, this address the first research question (1.3). As part of the further experiments, complex decision operations or state of deep-state machines are designed to have a higher edit distance between two graphs manipulated with two distinct operations.

In the second experiment (4.1.2) extended deep-state machines were trained to learn the graph construction sequences. To address the bias of DeepGG ([SG21]) or DGMG ([Li+18]) towards a specific graph traversal algorithm, a new model called GDTM is proposed. The GDTM allowed the deep-state machine to capture many alternative paths for traversing the same graph, enabling the model to learn diverse graph construction sequences during training. The results from the experiment showed that the deep-state machine was able to generate a graph construction sequence that could lead to a graph with $n = 83$ vertices in fewer steps. However, the generated graphs had a biased degree distribution towards a scale-free network and is unable to retain the structural similarity with the ground truth. Additionally, experiment (4.1.3) demonstrates the operation distribution per sequence location generated by the deep-state machine showed some similarity with ground truth. Overall, research suggests that the extended deep-state machine is able to learn and

generate sequences near to ground truth, but is not able to retain the structural similarity and degree distribution of graphs. One explanation for this can be the use of an unparameterized graph construction sequence that does not decide the connectivity of vertices and is dependent on the decision operation or state of the deep-state machine. Thus, this provides an answer to the second research question (1.3).

The extended deep-state machine is configured to have a larger number of states and higher complexity in each state compared with DeepGG or DGMG. In addition, experiment (4.1.1, 4.1.2, 4.1.3) showed some space of possibility to augment a deep-state machine with complex operations to some extent. Despite the extended deep-state machine being unable to learn the degree distributions of graphs, learning and generating graphs with an unparameterized graph construction sequence is illustrated by the experiments. The extended deep-state machine is designed to learn multiple alternatives of graph construction made possible by GDTM (3.3.3), hence all the above characteristics make it different from existing models.

6 Conclusion

In this research, novel concepts such as extended deep-state machines, intricate state or complex decision operations, a graph deconstruction tree model for generating graph representations as unparameterized construction sequences and techniques for learning complex graph embeddings and generating graphs using the trained model were introduced. The experiments illustrated the potential of augmenting the extended deep-state machine by integrating the above-mentioned concepts. The results showed that the auto-regressive models for graph generation can be extended to more states adding complexity to the states, compared to the existing model. To summarize, this research gives a new direction and techniques for advancing auto-regressive models for the graph generation.

To further improve the extended deep-state machine for graph generation, its limitations need to be addressed in the future. One important aspect is the extended deep-state machine should be capable of generating structurally similar graphs and degree distributions similar to ground truth. For this purpose, one approach would be to represent better connectivity of vertices through parameterized construction sequences or construction sequences that actually resemble the distribution of the target graphs. Another approach could be to not only learn the sequence of states but design the states to be intelligent enough to decide the vertex connectivity through the learning process. Furthermore, another approach can be based on deep reinforcement learning that involves sparse signals to acquire knowledge of construction sequences.

The aim of the research is to contribute insights for learning and generating graphs, as well as improve the extended deep-state machine. Finally, the future goal is to integrate the research into an end-to-end learning model that can be applied to various domains.

Bibliography

[Bar13] Albert-László Barabási. "Network science". In: *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* 371.1987 (2013), p. 20120375 (cit. on pp. 5, 6).

[BA99] Albert-László Barabási and Réka Albert. "Emergence of scaling in random networks". In: *science* 286.5439 (1999), pp. 509–512 (cit. on p. 25).

[BN06] Christopher M Bishop and Nasser M Nasrabadi. *Pattern recognition and machine learning*. Vol. 4. 4. Springer, 2006 (cit. on p. 11).

[Die06] R. Diestel. *Graph Theory*. Electronic library of mathematics. Springer, 2006. ISBN: 9783540261834. URL: `https://books.google.de/books?id=aR2TMYQr2CMC` (cit. on p. 5).

[Fla12] Peter Flach. *Machine learning: the art and science of algorithms that make sense of data*. Cambridge university press, 2012 (cit. on p. 21).

[Ham20] William L Hamilton. "Graph representation learning". In: *Synthesis Lectures on Artifical Intelligence and Machine Learning* 14.3 (2020), pp. 1–159 (cit. on pp. 4, 15, 16).

[Li+18] Yujia Li et al. "Learning deep generative models of graphs". In: *arXiv preprint arXiv:1803.03324* (2018) (cit. on pp. 2, 3, 15, 26, 39, 44, 48, 51, 72).

[Pav+11] Georgios A Pavlopoulos et al. "Using graph theory to analyze biological networks". In: *BioData mining* 4.1 (2011), pp. 1–27 (cit. on pp. 7, 8, 28, 29, 33–35).

[Rud16] Sebastian Ruder. "An overview of gradient descent optimization algorithms". In: *arXiv preprint arXiv:1609.04747* (2016) (cit. on pp. 11, 12).

[SG21] Julian Stier and Michael Granitzer. "DeepGG: A deep graph generator". In: *International Symposium on Intelligent Data Analysis*. Springer. 2021, pp. 313–324 (cit. on pp. 2, 3, 26, 39, 42, 44, 48, 50, 60, 72).

[TSS16] Ilya O Tolstikhin, Bharath K Sriperumbudur, and Bernhard Schölkopf. "Minimax estimation of maximum mean discrepancy with radial kernels". In: *Advances in Neural Information Processing Systems* 29 (2016) (cit. on p. 21).

[WS98] Duncan J Watts and Steven H Strogatz. "Collective dynamics of 'small-world'networks". In: *nature* 393.6684 (1998), pp. 440–442 (cit. on p. 25).

[Xu+18] Keyulu Xu et al. "How powerful are graph neural networks?" In: *arXiv preprint arXiv:1810.00826* (2018) (cit. on p. 15).

[YYL20] Jiaxuan You, Zhitao Ying, and Jure Leskovec. "Design space for graph neural networks". In: *Advances in Neural Information Processing Systems* 33 (2020) (cit. on pp. 1, 2).

[You+18] Jiaxuan You et al. "Graphrnn: Generating realistic graphs with deep auto-regressive models". In: *International conference on machine learning*. PMLR. 2018, pp. 5708–5717 (cit. on pp. 1, 2, 15, 26).

[ZL17] Junlong Zhang and Yu Luo. "Degree centrality, betweenness centrality, and closeness centrality in social network". In: *2017 2nd international conference on modelling, simulation and applied mathematics (MSAM2017)*. Atlantis Press. 2017, pp. 300–303 (cit. on p. 7).

[Zho+20] Jie Zhou et al. "Graph neural networks: A review of methods and applications". In: *AI Open* 1 (2020), pp. 57–81 (cit. on pp. 1, 14).

Eidesstattliche Erklärung

Hiermit versichere ich, dass ich diese Masterarbeit selbstständig und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel angefertigt habe und alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, als solche gekennzeichnet sind, sowie, dass ich die Masterarbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt habe.

Passau, April 18, 2023

_____

AUTHOR