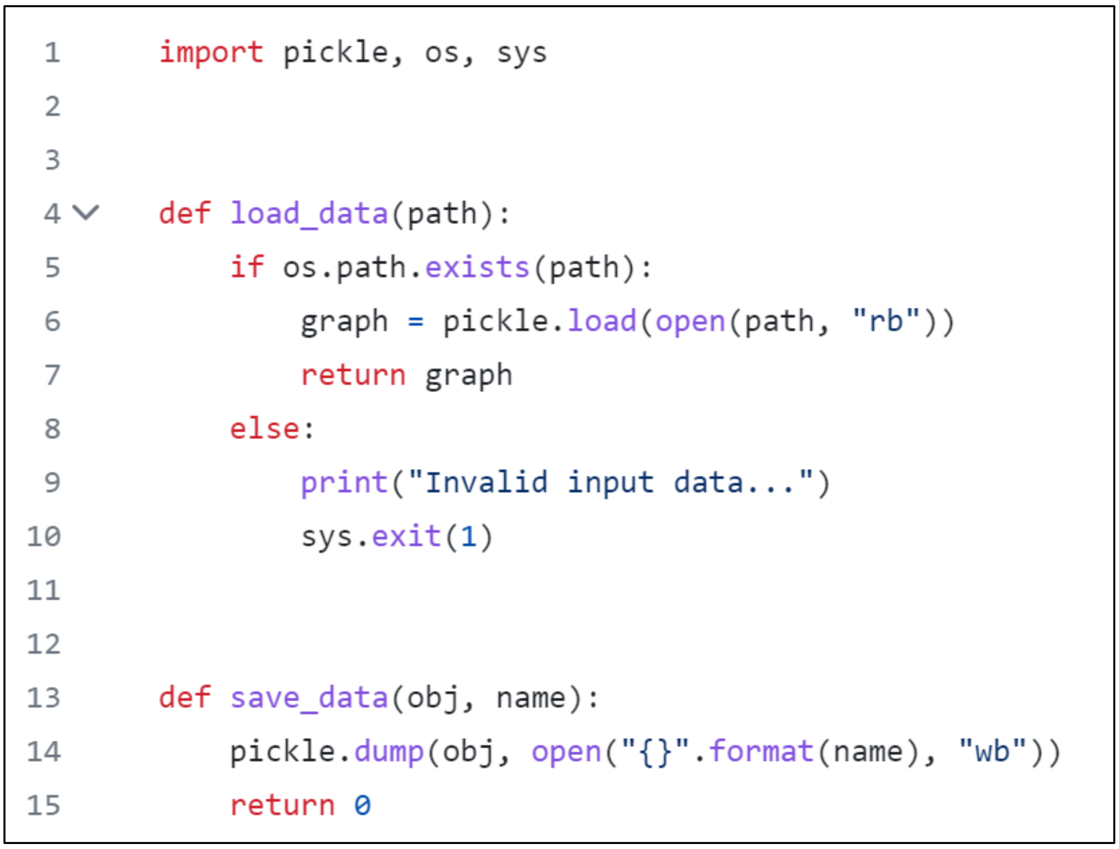
Tutorial for developer of GraphGenerator

This tutorial is for developer or researcher, which introduces the implemented functions of GraphGenerator and their specific features.

i) Preprocessing data

a) Data I/O operations (GraphGenerator/preprocessing/dataio.py)



```python
import pickle, os, sys


def load_data(path):
    if os.path.exists(path):
        graph = pickle.load(open(path, "rb"))
        return graph
    else:
        print("Invalid input data...")
        sys.exit(1)


def save_data(obj, name):
    pickle.dump(obj, open("{}".format(name), "wb"))
    return 0
```

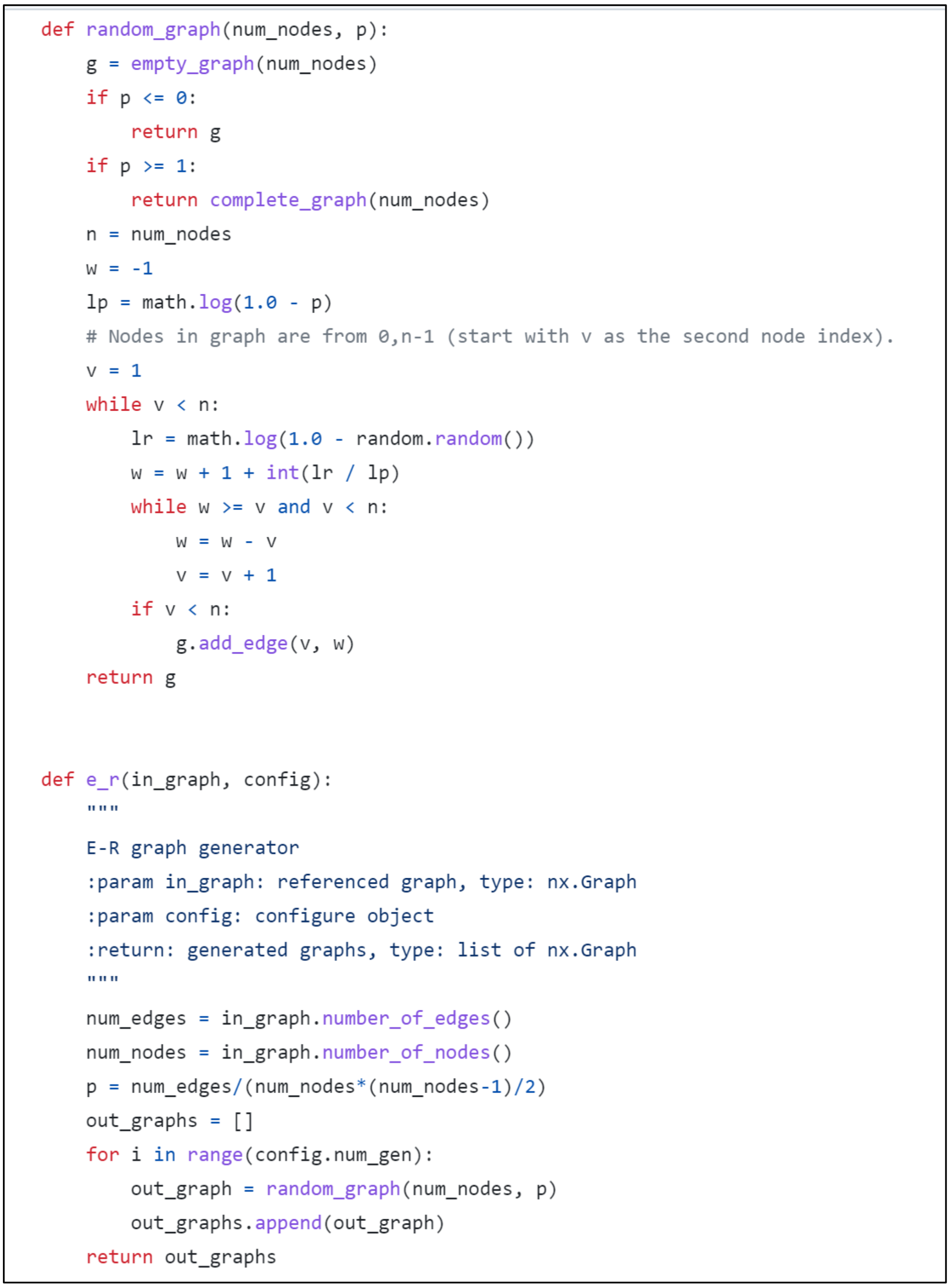
b) Other utils for data preprocessing (GraphGenerator/preprocess/utils.py)

```python
import networkx as nx
import sys


def edgelist_to_graph(path):
    try:
        graph = nx.read_edgelist(path)
        return graph
    except:
        print("Wrong path entered! Absolute path of edgelist file pxpected.")
        sys.exit(1)


def pathlist_to_graphlist(path):
    with open(path, "r") as f:
        path_list = f.readlines()
    path_list = [p.strip("\n") for p in path_list if p != "\n"]
    graph_list = [edgelist_to_graph(p) for p in path_list]
    return graph_list
```

ii) Generator models

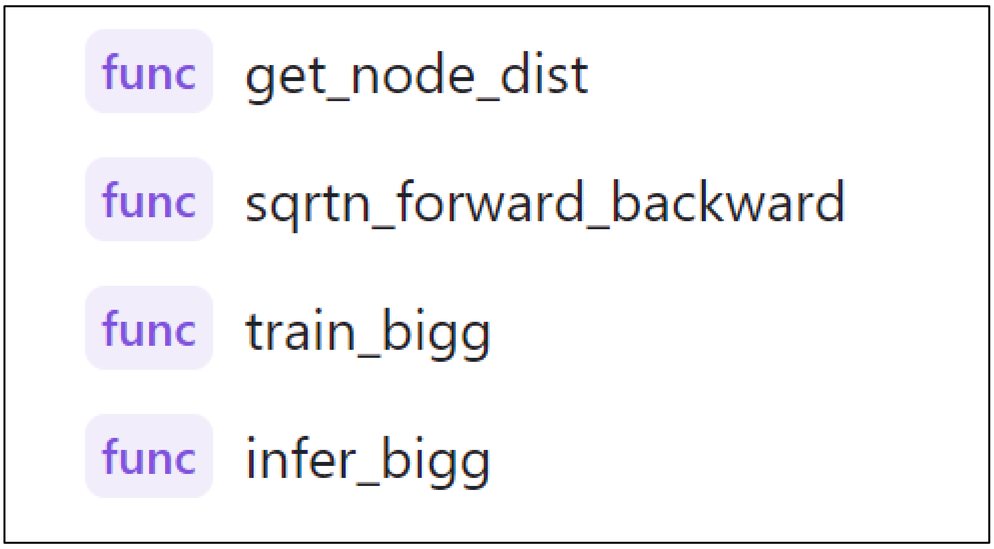
Each graph generator is implemented in an independent file, which is located in the 'models' folder. And if you want to copy the algorithm into your own machine, please refer to the specific file in the models folder. You can easily find the algorithm implementation function of your preferred graph generator.

a) Simple model-based graph generator: E-R (GraphGenerator/models/er.py)



```python
def random_graph(num_nodes, p):
    g = empty_graph(num_nodes)
    if p <= 0:
        return g
    if p >= 1:
        return complete_graph(num_nodes)
    n = num_nodes
    w = -1
    lp = math.log(1.0 - p)
    # Nodes in graph are from 0,n-1 (start with v as the second node index).
    v = 1
    while v < n:
        lr = math.log(1.0 - random.random())
        w = w + 1 + int(lr / lp)
        while w >= v and v < n:
            w = w - v
            v = v + 1
        if v < n:
            g.add_edge(v, w)
    return g


def e_r(in_graph, config):
    """
    E-R graph generator
    :param in_graph: referenced graph, type: nx.Graph
    :param config: configure object
    :return: generated graphs, type: list of nx.Graph
    """
    num_edges = in_graph.number_of_edges()
    num_nodes = in_graph.number_of_nodes()
    p = num_edges/(num_nodes*(num_nodes-1)/2)
    out_graphs = []
    for i in range(config.num_gen):
        out_graph = random_graph(num_nodes, p)
        out_graphs.append(out_graph)
    return out_graphs
```

b) Complex model-based graph generator: BiGG (GraphGenerator/models/bigg.py)
There are four functions of the complex model-based generator.



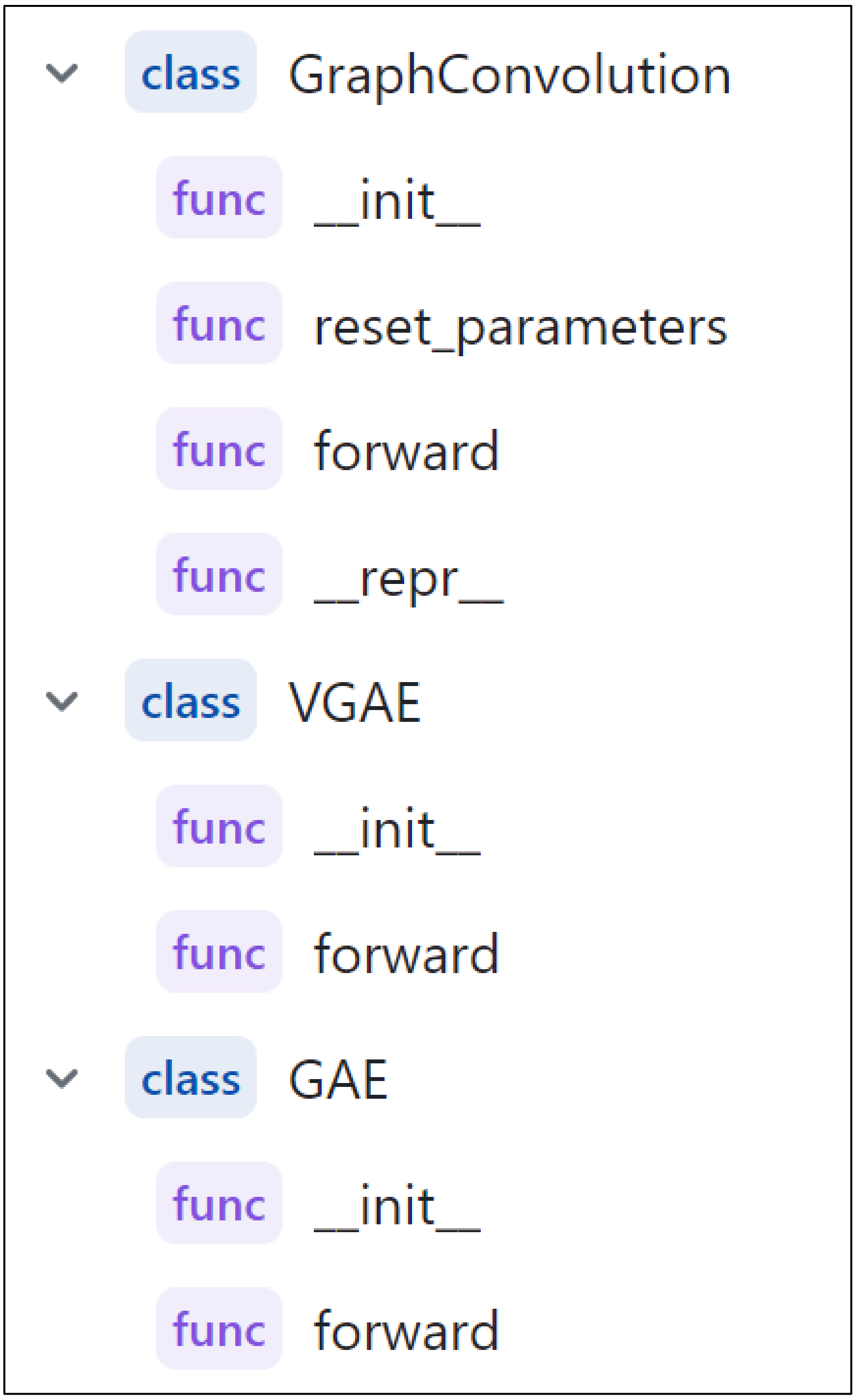
func get_node_dist

func sqrtn_forward_backward

func train_bigg

func infer_bigg

c) VAE-based graph generator: VGAE (GraphGenerator/models/vgae.py)
There are three classes to form the VGAE model.



```
class GraphConvolution
    func __init__
    func reset_parameters
    func forward
    func __repr__
class VGAE
    func __init__
    func forward
class GAE
    func __init__
    func forward
```

d) GAN-based graph generator: NetGAN (GraphGenerator/models/netgan.py)
Similar to VAE-based graph generator, you can find classes or functions about the detailed algorithm of your preferred model.