CS224W - Colab 5

In [1]:
```
!python -V
```

```
/bin/bash: /home/arch/anaconda3/envs/tf1.15_py3.8_gpu/lib/libtinfo.so.6: no v
ersion information available (required by /bin/bash)
Python 3.8.8
```

In [2]:
```
import os
```

In this Colab, we will shift our focus from homogenous graphs to heterogeneous graphs. Heterogeneous graphs extend the traditional homogenous graphs that we have been working with by incorperating different node and edge types. This additional information allows us to extend the graph neural nework models that we have worked with before. Namely, we can apply heterogenous message passing, where different message types now exist between different node and edge type relationships.

In this notebook, we will first learn how to transform NetworkX graphs into DeepSNAP representations. Then, we will dive deeper into how DeepSNAP stores and represents heterogeneous graphs as PyTorch Tensors.

Lastly, we will build our own heterogenous graph neural netowrk models using PyTorch Geometric and DeepSNAP. We will then apply our models for a node property prediction task; specifically, we will evaluate these models on the heterogeneous ACM node prediction dataset.

Note: Make sure to **sequentially run all the cells in each section**, so that the intermediate variables / packages will carry over to the next cell

Have fun and good luck on Colab 5 :)

Acknowledgement

Referenced the following workbook: https://notebooks.githubusercontent.com/view/ipynb?browser=chrome&color_mode=auto&commit=766b0c88a27d1f79817cec5578adf7c7c42bf7c2&devi

Device

You might need to use GPU for this Colab.

Please click `Runtime` and then `Change runtime type`. Then set the `hardware accelerator` to **GPU**.

Installation

In [3]:
```
!nvcc --version
```

```
/bin/bash: /home/arch/anaconda3/envs/tf1.15_py3.8_gpu/lib/libtinfo.so.6: no v
ersion information available (required by /bin/bash)
```

```
nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2021 NVIDIA Corporation
Built on Thu_Nov_18_09:45:30_PST_2021
Cuda compilation tools, release 11.5, V11.5.119
Build cuda_11.5.r11.5/compiler.30672275_0
```

In [4]:
```
import torch
print("pytorch version: {}".format(torch.__version__))
```

```
pytorch version: 1.10.2
```

In [61]:
```
import torch_geometric
torch_geometric.__version__
```

Out[61]:
```
'2.0.3'
```

In [84]:
```
!pip install torch-scatter -f https://data.pyg.ord/whl/torch-1.10.2+cu102.htm
!pip install torch-sparse -f https://data.pyg.ord/whl/torch-1.10.2+cu102.html
!pip install -q git+https://github.com/snap-stanford/deepsnap.git
!pip install -U -q PyDrive
```

```
/bin/bash: /home/arch/anaconda3/envs/tf1.15_py3.8_gpu/lib/libtinfo.so.6: no v
ersion information available (required by /bin/bash)
Looking in indexes: https://pypi.org/simple, https://pypi.ngc.nvidia.com
Looking in links: https://data.pyg.ord/whl/torch-1.10.2+cu102.html
Requirement already satisfied: torch-scatter in /home/arch/anaconda3/lib/pyth
on3.8/site-packages (2.0.7)
/bin/bash: /home/arch/anaconda3/envs/tf1.15_py3.8_gpu/lib/libtinfo.so.6: no v
ersion information available (required by /bin/bash)
Looking in indexes: https://pypi.org/simple, https://pypi.ngc.nvidia.com
Looking in links: https://data.pyg.ord/whl/torch-1.10.2+cu102.html
Requirement already satisfied: torch-sparse in /home/arch/anaconda3/lib/pytho
n3.8/site-packages (0.6.9)
Requirement already satisfied: scipy in /home/arch/anaconda3/lib/python3.8/si
te-packages (from torch-sparse) (1.6.2)
Requirement already satisfied: numpy<1.23.0,>=1.16.5 in /home/arch/anaconda3/
lib/python3.8/site-packages (from scipy->torch-sparse) (1.20.1)
/bin/bash: /home/arch/anaconda3/envs/tf1.15_py3.8_gpu/lib/libtinfo.so.6: no v
ersion information available (required by /bin/bash)
/bin/bash: /home/arch/anaconda3/envs/tf1.15_py3.8_gpu/lib/libtinfo.so.6: no v
ersion information available (required by /bin/bash)
```

In [85]:
```
# # Install torch geometric
# import os
# if 'IS_GRADESCOPE_ENV' not in os.environ:
#   !pip install torch-scatter -f https://data.pyg.org/whl/torch-1.10.0+cu111
#   !pip install torch-sparse -f https://data.pyg.org/whl/torch-1.10.0+cu111.
#   !pip install torch-geometric
#   !pip install -q git+https://github.com/snap-stanford/deepsnap.git
#   !pip install -U -q PyDrive
```

In [5]:
```
import os
if 'IS_GRADESCOPE_ENV' not in os.environ:
  !nvcc --version
  !python -c "import torch; print(torch.version.cuda)"
```

```
/bin/bash: /home/arch/anaconda3/envs/tf1.15_py3.8_gpu/lib/libtinfo.so.6: no v
ersion information available (required by /bin/bash)
nvcc: NVIDIA (R) Cuda compiler driver
Copyright (c) 2005-2021 NVIDIA Corporation
Built on Thu_Nov_18_09:45:30_PST_2021
Cuda compilation tools, release 11.5, V11.5.119
```

```
Build cuda_11.5.r11.5/compiler.30672275_0
/bin/bash: /home/arch/anaconda3/envs/tf1.15_py3.8_gpu/lib/libtinfo.so.6: no v
ersion information available (required by /bin/bash)
10.2
```

In [6]:

```
!python -V
```

```
/bin/bash: /home/arch/anaconda3/envs/tf1.15_py3.8_gpu/lib/libtinfo.so.6: no v
ersion information available (required by /bin/bash)
Python 3.8.8
```

In [7]:

```
if 'IS_GRADESCOPE_ENV' not in os.environ:
  import torch
  print("pytorch version: {}".format(torch.__version__))
  import torch_geometric
  print("torch geometric version: {}".format(torch_geometric.__version__))
```

```
pytorch version: 1.10.2
/home/arch/anaconda3/envs/GNN_env/lib/python3.8/site-packages/torch/cuda/__in
it__.py:80: UserWarning: CUDA initialization: CUDA unknown error - this may b
e due to an incorrectly set up environment, e.g. changing env variable CUDA_V
ISIBLE_DEVICES after program start. Setting the available devices to be zero.
(Triggered internally at  /opt/conda/conda-bld/pytorch_1640811757556/work/c1
0/cuda/CUDAFunctions.cpp:112.)
  return torch._C._cuda_getDeviceCount() > 0
torch geometric version: 2.0.3
```

DeepSNAP Basics

In previous Colabs we used both of graph class (NetworkX) and tensor (PyG) representations of graphs separately. The graph class `nx.Graph` provides rich analysis and manipulation functionalities, such as the clustering coefficient and PageRank. To feed the graph into the model, we need to transform the graph into tensor representations including edge tensor `edge_index` and node attributes tensors `x` and `y`. But only using tensors (as the graphs formatted in PyG `datasets` and `data`) will make many graph manipulations and analysis less efficient and harder. So, in this Colab we will use DeepSNAP which combines both representations and offers a full pipeline for GNN training / validation / testing.

In general, DeepSNAP is a Python library to assist efficient deep learning on graphs. DeepSNAP features in its support for flexible graph manipulation, standard pipeline, heterogeneous graphs and simple API.

1. DeepSNAP is easy to be used for the sophisticated graph manipulations, such as feature computation, pretraining, subgraph extraction etc. during/before the training.
2. In most frameworks, standard pipelines for node, edge, link, graph-level tasks under inductive or transductive settings are left to the user to code. In practice, there are additional design choices involved (such as how to split dataset for link prediction). DeepSNAP provides such a standard pipeline that greatly saves repetitive coding efforts, and enables fair comparision for models.
3. Many real-world graphs are heterogeneous graphs. But packages support for heterogeneous graphs, including data storage and flexible message passing, is lacking. DeepSNAP provides an efficient and flexible heterogeneous graph that supports both the node and edge heterogeneity.

DeepSNAP is a newly released project and it is still under development. If you find any bugs or have any improvement ideas, feel free to raise issues or create pull requests on the GitHub directly :)

In this Colab, we will focus on learning using Heterogeneous Graphs. Not many libraries are able to handle heterogeneous graphs, but DeepSNAP handles them quite elegantly, which is why we're introducing it here!

1) DeepSNAP Heterogeneous Graph

First, we will explore how to transform a NetworkX graph into the format supported by DeepSNAP.

In DeepSNAP we have three levels of attributes. We can have **node level** attributes including `node_feature` and `node_label`. The other two levels of attributes are graph and edge attributes. The usage is similar to the node level one except that the feature becomes `edge_feature` or `graph_feature` and label becomes `edge_label` or `graph_label` etc.

DeepSNAP extends its traditional graph representation to include heterogeneous graphs by including the following graph property features:

- `node_feature` : The feature of each node (`torch.tensor`)
- `edge_feature` : The feature of each edge (`torch.tensor`)
- `node_label` : The label of each node (`int`)
- `node_type` : The node type of each node (`string`)
- `edge_type` : The edge type of each edge (`string`)

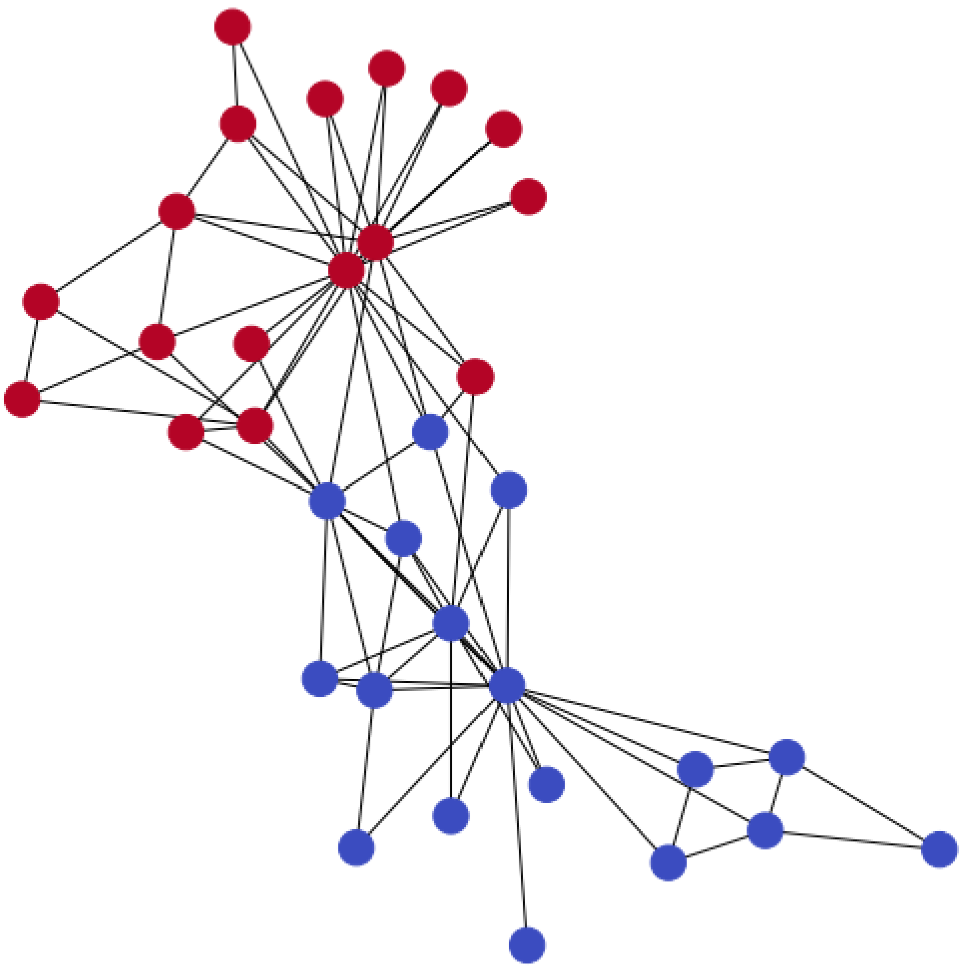
where the key **new** features we add are `node_type` and `edge_type`, which enables us to perform heterogenous message passing.

For this first question we will work with the familiar karate club graph seen in Colab 1. To start, since each node in the graph belongs to one of two clubs (club "Mr. Hi" or club "Officer"), we will treat the club as the `node_type`. The code below demonstrates how to differentiate the nodes in the NetworkX graph.

In [66]:

```
import networkx as nx
from networkx.algorithms.community import greedy_modularity_communities
import matplotlib.pyplot as plt
import copy

if 'IS_GRADESCOPE_ENV' not in os.environ:
  from pylab import show
  G = nx.karate_club_graph()
  community_map = {}
  for node in G.nodes(data=True):
    if node[1]["club"] == "Mr. Hi":
      community_map[node[0]] = 0
    else:
      community_map[node[0]] = 1
  node_color = []
  color_map = {0: 0, 1: 1}
  node_color = [color_map[community_map[node]] for node in G.nodes()]
```

```
pos = nx.spring_layout(G)
plt.figure(figsize=(7, 7))
nx.draw(G, pos=pos, cmap=plt.get_cmap('coolwarm'), node_color=node_color)
show()
```



Question 1.1: Assigning Node Type and Node Features

Using the `community_map` dictionary and graph `G` from above, add node attributes `node_type` and `node_label` to the graph G. Namely, for `node_type` assign nodes in the "Mr. Hi" club to a node type `n0` and nodes in club "Officer" a node type `n1` . Note: the node type should be a `string` property.

Then for `node_label` , assign nodes in "Mr. Hi" club to a `node_label` `0` and nodes in club "Officer" a `node_label` of `1` .

Lastly, assign every node the *tensor* feature vector $[1, 1, 1, 1, 1]$.

Hint: Look at the NetworkX function `nx.classes.function.set_node_attributes` .

Note: This question is not specifically graded but is important for later questions.

In [9]:

```
import torch

def assign_node_types(G, community_map):
  # TODO: Implement a function that takes in a NetworkX graph
  # G and community map assignment (mapping node id --> 0/1 label)
  # and adds 'node_type' as a node_attribute in G.
```

```
    ############# Your code here ############
    ## (~2 line of code)
    ## Note
    ## 1. Look up NetworkX `nx.classes.function.set_node_attributes`
    ## 2. Look above for the two node type values!
      node_type_map = {0:'n0', 1:'n1'}
      node_types = {node:node_type_map[community_map[node]] for node in G.nodes
      nx.set_node_attributes(G, values=node_types, name="node_type")


    #########################################

  def assign_node_labels(G, community_map):
    # TODO: Implement a function that takes in a NetworkX graph
    # G and community map assignment (mapping node id --> 0/1 label)
    # and adds 'node_label' as a node_attribute in G.

    ############# Your code here ############
    ## (~2 line of code)
    ## Note
    ## 1. Look up NetworkX `nx.classes.function.set_node_attributes`
      nx.set_node_attributes(G, values=community_map, name="node_label")
      pass

    #########################################

  def assign_node_features(G):
    # TODO: Implement a function that takes in a NetworkX graph
    # G and adds 'node_feature' as a node_attribute in G. Each node
    # in the graph has the same feature vector - a torchtensor with
    # data [1., 1., 1., 1., 1.]

    ############# Your code here ############
    ## (~2 line of code)
    ## Note
    ## 1. Look up NetworkX `nx.classes.function.set_node_attributes`
      torch_tensor = torch.ones(5)
      nx.set_node_attributes(G, values=torch_tensor, name="node_feature")


    #########################################

  if 'IS_GRADESCOPE_ENV' not in os.environ:
    assign_node_types(G, community_map)
    assign_node_labels(G, community_map)
    assign_node_features(G)

    # Explore node properties for the node with id: 20
    node_id = 20
    print (f"Node {node_id} has properties:", G.nodes(data=True)[node_id])

    node_id = 0
    print (f"Node {node_id} has properties:", G.nodes(data=True)[node_id])
```

```
Node 20 has properties: {'club': 'Officer', 'node_type': 'n1', 'node_label':
1, 'node_feature': tensor([1., 1., 1., 1., 1.])}
Node 0 has properties: {'club': 'Mr. Hi', 'node_type': 'n0', 'node_label': 0,
'node_feature': tensor([1., 1., 1., 1., 1.])}
```

Question 1.2: Assigning Edge Types

Next, we will assign three different `edge_types` :

- Edges within club "Mr. Hi": `e0`
- Edges within club "Officer": `e1`
- Edges between the two clubs: `e2`

Hint: Use the `community_map` from before and `nx.classes.function.set_edge_attributes`

In [10]:

```
G.edges
```

Out[10]:

```
EdgeView([(0, 1), (0, 2), (0, 3), (0, 4), (0, 5), (0, 6), (0, 7), (0, 8), (0, 10), (0, 11), (0, 12), (0, 13), (0, 17), (0, 19), (0, 21), (0, 31), (1, 2), (1, 3), (1, 7), (1, 13), (1, 17), (1, 19), (1, 21), (1, 30), (2, 3), (2, 7), (2, 8), (2, 9), (2, 13), (2, 27), (2, 28), (2, 32), (3, 7), (3, 12), (3, 13), (4, 6), (4, 10), (5, 6), (5, 10), (5, 16), (6, 16), (8, 30), (8, 32), (8, 33), (9, 33), (13, 33), (14, 32), (14, 33), (15, 32), (15, 33), (18, 32), (18, 33), (19, 33), (20, 32), (20, 33), (22, 32), (22, 33), (23, 25), (23, 27), (23, 29), (23, 32), (23, 33), (24, 25), (24, 27), (24, 31), (25, 31), (26, 29), (26, 33), (27, 33), (28, 31), (28, 33), (29, 32), (29, 33), (30, 32), (30, 33), (31, 32), (31, 33), (32, 33)])
```

In [11]:

```python
def get_edge_type(u,v,G):
#    node1 = edge[0]
#    node2 = edge[1]

    club1 = G.nodes(data=True)[u]['club']
    club2 = G.nodes(data=True)[v]['club']

    if (club1=="Mr. Hi" and club2=="Mr. Hi"):
        edge_type = 'e0'
    elif (club1=="Officer" and club2=="Officer"):
        edge_type = 'e1'
    else:
        edge_type = 'e2'
    return edge_type

attr = {(u,v):{"edge_type": get_edge_type(u,v,G)} for (u,v) in G.edges()}
print(attr)
```

```
{(0, 1): {'edge_type': 'e0'}, (0, 2): {'edge_type': 'e0'}, (0, 3): {'edge_type': 'e0'}, (0, 4): {'edge_type': 'e0'}, (0, 5): {'edge_type': 'e0'}, (0, 6): {'edge_type': 'e0'}, (0, 7): {'edge_type': 'e0'}, (0, 8): {'edge_type': 'e0'}, (0, 10): {'edge_type': 'e0'}, (0, 11): {'edge_type': 'e0'}, (0, 12): {'edge_type': 'e0'}, (0, 13): {'edge_type': 'e0'}, (0, 17): {'edge_type': 'e0'}, (0, 19): {'edge_type': 'e0'}, (0, 21): {'edge_type': 'e0'}, (0, 31): {'edge_type': 'e2'}, (1, 2): {'edge_type': 'e0'}, (1, 3): {'edge_type': 'e0'}, (1, 7): {'edge_type': 'e0'}, (1, 13): {'edge_type': 'e0'}, (1, 17): {'edge_type': 'e0'}, (1, 19): {'edge_type': 'e0'}, (1, 21): {'edge_type': 'e0'}, (1, 30): {'edge_type': 'e2'}, (2, 3): {'edge_type': 'e0'}, (2, 7): {'edge_type': 'e0'}, (2, 8): {'edge_type': 'e0'}, (2, 9): {'edge_type': 'e2'}, (2, 13): {'edge_type': 'e0'}, (2, 27): {'edge_type': 'e2'}, (2, 28): {'edge_type': 'e2'}, (2, 32): {'edge_type': 'e2'}, (3, 7): {'edge_type': 'e0'}, (3, 12): {'edge_type': 'e0'}, (3, 13): {'edge_type': 'e0'}, (4, 6): {'edge_type': 'e0'}, (4, 10): {'edge_type': 'e0'}, (5, 6): {'edge_type': 'e0'}, (5, 10): {'edge_type': 'e0'}, (5, 16): {'edge_type': 'e0'}, (6, 16): {'edge_type': 'e0'}, (8, 30): {'edge_type': 'e2'}, (8, 32): {'edge_type': 'e2'}, (8, 33): {'edge_type': 'e2'}, (9, 33): {'edge_type': 'e1'}, (13, 33): {'edge_type': 'e2'}, (14, 32): {'edge_type': 'e1'}, (14, 33): {'edge_type': 'e1'}, (15, 32): {'edge_type': 'e1'}, (15, 33): {'edge_type': 'e1'}, (18, 32): {'edge_type': 'e1'}, (18, 33): {'edge_type': 'e1'}, (19, 33): {'edge_type': 'e2'}, (20, 32): {'edge_type': 'e1'}, (20, 33): {'edge_type': 'e1'}, (22, 32): {'edge_type': 'e1'}, (22, 33): {'edge_type': 'e1'}, (23, 25): {'edge_type': 'e1'}, (23, 27): {'edge_type': 'e1'}, (23, 29): {'edge_type': 'e1'}, (23, 32): {'edge_type': 'e1'}, (23, 33): {'edge_type': 'e1'}, (24, 25): {'edge_type': 'e1'}, (24, 27): {'edge_typ
```

```
e': 'e1'}, (24, 31): {'edge_type': 'e1'}, (25, 31): {'edge_type': 'e1'}, (26,
29): {'edge_type': 'e1'}, (26, 33): {'edge_type': 'e1'}, (27, 33): {'edge_typ
e': 'e1'}, (28, 31): {'edge_type': 'e1'}, (28, 33): {'edge_type': 'e1'}, (29,
32): {'edge_type': 'e1'}, (29, 33): {'edge_type': 'e1'}, (30, 32): {'edge_typ
e': 'e1'}, (30, 33): {'edge_type': 'e1'}, (31, 32): {'edge_type': 'e1'}, (31,
33): {'edge_type': 'e1'}, (32, 33): {'edge_type': 'e1'}}
```

In [12]:

```python
edge_dict={"edge":[],"edge_type":[]}

for edge in G.edges():
    node1 = edge[0]
    node2 = edge[1]

    club1 = G.nodes(data=True)[node1]['club']
    club2 = G.nodes(data=True)[node2]['club']

    if (club1=="Mr. Hi" and club2=="Mr. Hi"):
        edge_type = 0
    elif (club1=="Officer" and club2=="Officer"):
        edge_type = 1
    else:
        edge_type = 2

    edge_dict["edge"].append(edge)
    edge_dict["edge_type"].append(edge_type)

    print("Edge:{} comprising node1:{} club1:{} and node2:{} club2:{} ---> ec

#attrs = {"Mr. Hi":e0, "Officer":e1}
l = list(G.edges())
print(edge_dict)
```

```
Edge:(0, 1) comprising node1:0 club1:Mr. Hi and node2:1 club2:Mr. Hi ---> edg
e_type:0
Edge:(0, 2) comprising node1:0 club1:Mr. Hi and node2:2 club2:Mr. Hi ---> edg
e_type:0
Edge:(0, 3) comprising node1:0 club1:Mr. Hi and node2:3 club2:Mr. Hi ---> edg
e_type:0
Edge:(0, 4) comprising node1:0 club1:Mr. Hi and node2:4 club2:Mr. Hi ---> edg
e_type:0
Edge:(0, 5) comprising node1:0 club1:Mr. Hi and node2:5 club2:Mr. Hi ---> edg
e_type:0
Edge:(0, 6) comprising node1:0 club1:Mr. Hi and node2:6 club2:Mr. Hi ---> edg
e_type:0
Edge:(0, 7) comprising node1:0 club1:Mr. Hi and node2:7 club2:Mr. Hi ---> edg
e_type:0
Edge:(0, 8) comprising node1:0 club1:Mr. Hi and node2:8 club2:Mr. Hi ---> edg
e_type:0
Edge:(0, 10) comprising node1:0 club1:Mr. Hi and node2:10 club2:Mr. Hi ---> e
dge_type:0
Edge:(0, 11) comprising node1:0 club1:Mr. Hi and node2:11 club2:Mr. Hi ---> e
dge_type:0
Edge:(0, 12) comprising node1:0 club1:Mr. Hi and node2:12 club2:Mr. Hi ---> e
dge_type:0
Edge:(0, 13) comprising node1:0 club1:Mr. Hi and node2:13 club2:Mr. Hi ---> e
dge_type:0
Edge:(0, 17) comprising node1:0 club1:Mr. Hi and node2:17 club2:Mr. Hi ---> e
dge_type:0
Edge:(0, 19) comprising node1:0 club1:Mr. Hi and node2:19 club2:Mr. Hi ---> e
dge_type:0
Edge:(0, 21) comprising node1:0 club1:Mr. Hi and node2:21 club2:Mr. Hi ---> e
dge_type:0
Edge:(0, 31) comprising node1:0 club1:Mr. Hi and node2:31 club2:Officer --->
edge_type:2
Edge:(1, 2) comprising node1:1 club1:Mr. Hi and node2:2 club2:Mr. Hi ---> edg
e_type:0
```

```
Edge:(1, 3) comprising node1:1 club1:Mr. Hi and node2:3 club2:Mr. Hi ---> edg
e_type:0
Edge:(1, 7) comprising node1:1 club1:Mr. Hi and node2:7 club2:Mr. Hi ---> edg
e_type:0
Edge:(1, 13) comprising node1:1 club1:Mr. Hi and node2:13 club2:Mr. Hi ---> e
dge_type:0
Edge:(1, 17) comprising node1:1 club1:Mr. Hi and node2:17 club2:Mr. Hi ---> e
dge_type:0
Edge:(1, 19) comprising node1:1 club1:Mr. Hi and node2:19 club2:Mr. Hi ---> e
dge_type:0
Edge:(1, 21) comprising node1:1 club1:Mr. Hi and node2:21 club2:Mr. Hi ---> e
dge_type:0
Edge:(1, 30) comprising node1:1 club1:Mr. Hi and node2:30 club2:Officer --->
edge_type:2
Edge:(2, 3) comprising node1:2 club1:Mr. Hi and node2:3 club2:Mr. Hi ---> edg
e_type:0
Edge:(2, 7) comprising node1:2 club1:Mr. Hi and node2:7 club2:Mr. Hi ---> edg
e_type:0
Edge:(2, 8) comprising node1:2 club1:Mr. Hi and node2:8 club2:Mr. Hi ---> edg
e_type:0
Edge:(2, 9) comprising node1:2 club1:Mr. Hi and node2:9 club2:Officer ---> ed
ge_type:2
Edge:(2, 13) comprising node1:2 club1:Mr. Hi and node2:13 club2:Mr. Hi ---> e
dge_type:0
Edge:(2, 27) comprising node1:2 club1:Mr. Hi and node2:27 club2:Officer --->
edge_type:2
Edge:(2, 28) comprising node1:2 club1:Mr. Hi and node2:28 club2:Officer --->
edge_type:2
Edge:(2, 32) comprising node1:2 club1:Mr. Hi and node2:32 club2:Officer --->
edge_type:2
Edge:(3, 7) comprising node1:3 club1:Mr. Hi and node2:7 club2:Mr. Hi ---> edg
e_type:0
Edge:(3, 12) comprising node1:3 club1:Mr. Hi and node2:12 club2:Mr. Hi ---> e
dge_type:0
Edge:(3, 13) comprising node1:3 club1:Mr. Hi and node2:13 club2:Mr. Hi ---> e
dge_type:0
Edge:(4, 6) comprising node1:4 club1:Mr. Hi and node2:6 club2:Mr. Hi ---> edg
e_type:0
Edge:(4, 10) comprising node1:4 club1:Mr. Hi and node2:10 club2:Mr. Hi ---> e
dge_type:0
Edge:(5, 6) comprising node1:5 club1:Mr. Hi and node2:6 club2:Mr. Hi ---> edg
e_type:0
Edge:(5, 10) comprising node1:5 club1:Mr. Hi and node2:10 club2:Mr. Hi ---> e
dge_type:0
Edge:(5, 16) comprising node1:5 club1:Mr. Hi and node2:16 club2:Mr. Hi ---> e
dge_type:0
Edge:(6, 16) comprising node1:6 club1:Mr. Hi and node2:16 club2:Mr. Hi ---> e
dge_type:0
Edge:(8, 30) comprising node1:8 club1:Mr. Hi and node2:30 club2:Officer --->
edge_type:2
Edge:(8, 32) comprising node1:8 club1:Mr. Hi and node2:32 club2:Officer --->
edge_type:2
Edge:(8, 33) comprising node1:8 club1:Mr. Hi and node2:33 club2:Officer --->
edge_type:2
Edge:(9, 33) comprising node1:9 club1:Officer and node2:33 club2:Officer --->
edge_type:1
Edge:(13, 33) comprising node1:13 club1:Mr. Hi and node2:33 club2:Officer ---
> edge_type:2
Edge:(14, 32) comprising node1:14 club1:Officer and node2:32 club2:Officer --
-> edge_type:1
Edge:(14, 33) comprising node1:14 club1:Officer and node2:33 club2:Officer --
-> edge_type:1
Edge:(15, 32) comprising node1:15 club1:Officer and node2:32 club2:Officer --
-> edge_type:1
Edge:(15, 33) comprising node1:15 club1:Officer and node2:33 club2:Officer --
-> edge_type:1
Edge:(18, 32) comprising node1:18 club1:Officer and node2:32 club2:Officer --
-> edge_type:1
Edge:(18, 33) comprising node1:18 club1:Officer and node2:33 club2:Officer --
```

```
-> edge_type:1
Edge:(19, 33) comprising node1:19 club1:Mr. Hi and node2:33 club2:Officer ---
> edge_type:2
Edge:(20, 32) comprising node1:20 club1:Officer and node2:32 club2:Officer --
-> edge_type:1
Edge:(20, 33) comprising node1:20 club1:Officer and node2:33 club2:Officer --
-> edge_type:1
Edge:(22, 32) comprising node1:22 club1:Officer and node2:32 club2:Officer --
-> edge_type:1
Edge:(22, 33) comprising node1:22 club1:Officer and node2:33 club2:Officer --
-> edge_type:1
Edge:(23, 25) comprising node1:23 club1:Officer and node2:25 club2:Officer --
-> edge_type:1
Edge:(23, 27) comprising node1:23 club1:Officer and node2:27 club2:Officer --
-> edge_type:1
Edge:(23, 29) comprising node1:23 club1:Officer and node2:29 club2:Officer --
-> edge_type:1
Edge:(23, 32) comprising node1:23 club1:Officer and node2:32 club2:Officer --
-> edge_type:1
Edge:(23, 33) comprising node1:23 club1:Officer and node2:33 club2:Officer --
-> edge_type:1
Edge:(24, 25) comprising node1:24 club1:Officer and node2:25 club2:Officer --
-> edge_type:1
Edge:(24, 27) comprising node1:24 club1:Officer and node2:27 club2:Officer --
-> edge_type:1
Edge:(24, 31) comprising node1:24 club1:Officer and node2:31 club2:Officer --
-> edge_type:1
Edge:(25, 31) comprising node1:25 club1:Officer and node2:31 club2:Officer --
-> edge_type:1
Edge:(26, 29) comprising node1:26 club1:Officer and node2:29 club2:Officer --
-> edge_type:1
Edge:(26, 33) comprising node1:26 club1:Officer and node2:33 club2:Officer --
-> edge_type:1
Edge:(27, 33) comprising node1:27 club1:Officer and node2:33 club2:Officer --
-> edge_type:1
Edge:(28, 31) comprising node1:28 club1:Officer and node2:31 club2:Officer --
-> edge_type:1
Edge:(28, 33) comprising node1:28 club1:Officer and node2:33 club2:Officer --
-> edge_type:1
Edge:(29, 32) comprising node1:29 club1:Officer and node2:32 club2:Officer --
-> edge_type:1
Edge:(29, 33) comprising node1:29 club1:Officer and node2:33 club2:Officer --
-> edge_type:1
Edge:(30, 32) comprising node1:30 club1:Officer and node2:32 club2:Officer --
-> edge_type:1
Edge:(30, 33) comprising node1:30 club1:Officer and node2:33 club2:Officer --
-> edge_type:1
Edge:(31, 32) comprising node1:31 club1:Officer and node2:32 club2:Officer --
-> edge_type:1
Edge:(31, 33) comprising node1:31 club1:Officer and node2:33 club2:Officer --
-> edge_type:1
Edge:(32, 33) comprising node1:32 club1:Officer and node2:33 club2:Officer --
-> edge_type:1
{'edge': [(0, 1), (0, 2), (0, 3), (0, 4), (0, 5), (0, 6), (0, 7), (0, 8), (0,
10), (0, 11), (0, 12), (0, 13), (0, 17), (0, 19), (0, 21), (0, 31), (1, 2),
(1, 3), (1, 7), (1, 13), (1, 17), (1, 19), (1, 21), (1, 30), (2, 3), (2, 7),
(2, 8), (2, 9), (2, 13), (2, 27), (2, 28), (2, 32), (3, 7), (3, 12), (3, 13),
(4, 6), (4, 10), (5, 6), (5, 10), (5, 16), (6, 16), (8, 30), (8, 32), (8, 3
3), (9, 33), (13, 33), (14, 32), (14, 33), (15, 32), (15, 33), (18, 32), (18,
33), (19, 33), (20, 32), (20, 33), (22, 32), (22, 33), (23, 25), (23, 27), (2
3, 29), (23, 32), (23, 33), (24, 25), (24, 27), (24, 31), (25, 31), (26, 29),
(26, 33), (27, 33), (28, 31), (28, 33), (29, 32), (29, 33), (30, 32), (30, 3
3), (31, 32), (31, 33), (32, 33)], 'edge_type': [0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 2, 0, 0, 0, 0, 0, 0, 0, 2, 0, 0, 0, 2, 0, 2, 2, 2, 0, 0, 0,
0, 0, 0, 0, 0, 0, 2, 2, 2, 1, 2, 1, 1, 1, 1, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]}
```

In [13]:

```python
def assign_edge_types(G, community_map):
```

```
  # TODO: Implement a function that takes in a NetworkX graph
  # G and community map assignment (mapping node id --> 0/1 label)
  # and adds 'edge_type' as a edge_attribute in G.

  ############# Your code here ############
  ## (~5 line of code)
  ## Note
  ## 1. Create an edge assignment dict following rules above
  ## 2. Look up NetworkX `nx.classes.function.set_edge_attributes`
    def get_edge_type(u,v,G):

        club1 = G.nodes(data=True)[u]['club']
        club2 = G.nodes(data=True)[v]['club']

        if (club1=="Mr. Hi" and club2=="Mr. Hi"):
            edge_type = 'e0'
        elif (club1=="Officer" and club2=="Officer"):
            edge_type = 'e1'
        else:
            edge_type = 'e2'
        return edge_type


    attr = {(u,v):{"edge_type": get_edge_type(u,v,G)} for (u,v) in G.edges()}

    #edge_dict["edge"].append(edge)
    #edge_dict["edge_type"].append(edge_type)

    nx.set_edge_attributes(G,attr)
    pass

  #########################################

if 'IS_GRADESCOPE_ENV' not in os.environ:
  assign_edge_types(G, community_map)

  # Explore edge properties for a sampled edge and check the corresponding
  # node types
  edge_idx = 15
  n1 = 0
  n2 = 31
  edge = list(G.edges(data=True))[edge_idx]
  print (f"Edge ({edge[0]}, {edge[1]}) has properties:", edge[2])
  print (f"Node {n1} has properties:", G.nodes(data=True)[n1])
  print (f"Node {n2} has properties:", G.nodes(data=True)[n2])
```

```
Edge (0, 31) has properties: {'edge_type': 'e2'}
Node 0 has properties: {'club': 'Mr. Hi', 'node_type': 'n0', 'node_label': 0,
'node_feature': tensor([1., 1., 1., 1., 1.])}
Node 31 has properties: {'club': 'Officer', 'node_type': 'n1', 'node_label':
1, 'node_feature': tensor([1., 1., 1., 1., 1.])}
```

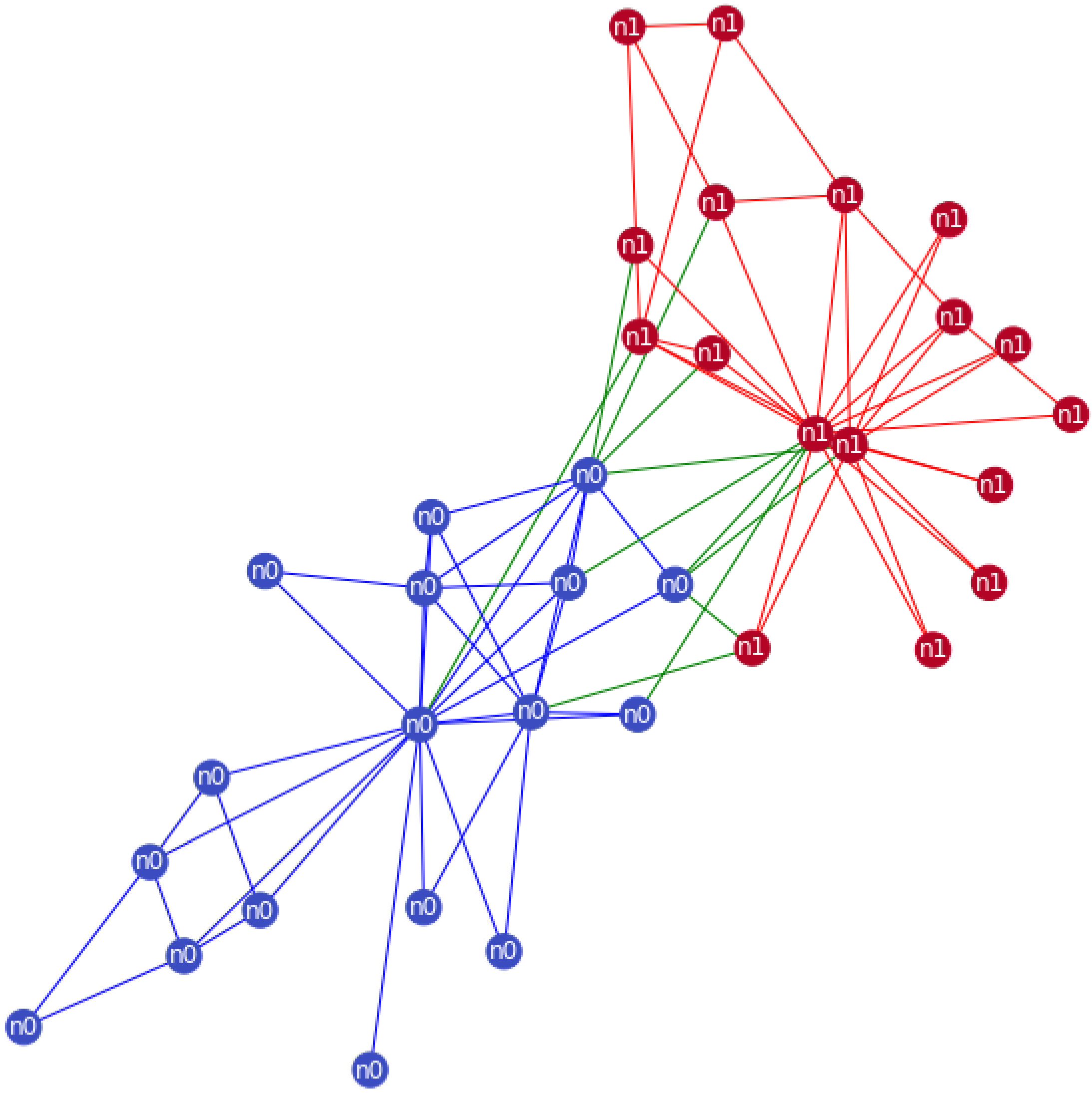
Heterogeneous Graph Visualization

Now we can visualize the Heterogeneous Graph we have generated.

In [14]:

```
if 'IS_GRADESCOPE_ENV' not in os.environ:
  edge_color = {}
  for edge in G.edges():
    n1, n2 = edge
    edge_color[edge] = community_map[n1] if community_map[n1] == community_ma
    if community_map[n1] == community_map[n2] and community_map[n1] == 0:
```

```
      edge_color[edge] = 'blue'
    elif community_map[n1] == community_map[n2] and community_map[n1] == 1:
      edge_color[edge] = 'red'
    else:
      edge_color[edge] = 'green'

  G_orig = copy.deepcopy(G)
  nx.classes.function.set_edge_attributes(G, edge_color, name='color')
  colors = nx.get_edge_attributes(G,'color').values()
  labels = nx.get_node_attributes(G, 'node_type')
  plt.figure(figsize=(8, 8))
  nx.draw(G, pos=pos, cmap=plt.get_cmap('coolwarm'), node_color=node_color, e
  show()
```



where we differentiate edges within each clubs (2 types) and edges between the two clubs (1 type). Different types of nodes and edges are visualized in different colors. The NetworkX object `G` in following code can be transformed into `deepsnap.hetero_graph.HeteroGraph` directly.

Transforming to DeepSNAP representation

We will now work through transforming the NetworkX object `G` into a `deepsnap.hetero_graph.HeteroGraph`.

In [15]:
```
from deepsnap.hetero_graph import HeteroGraph

if 'IS_GRADESCOPE_ENV' not in os.environ:
  hete = HeteroGraph(G_orig)
```

Question 1.3: How many nodes are of each type (10 Points)

In [16]:
```
num_nodes=hete.num_nodes()
print("num_nodes:{}".format(num_nodes))
num_nodes['n0']
```

```
num_nodes:{'n0': 17, 'n1': 17}
```

Out[16]: 17

In [17]:
```
def get_nodes_per_type(hete):
  # TODO: Implement a function that takes a DeepSNAP dataset object
  # and return the number of nodes per `node_type`.

    num_nodes_n0 = 0
    num_nodes_n1 = 0

  ############# Your code here ############
  ## (~2 line of code)
  ## Note
  ## 1. Colab autocomplete functionality might be useful.
    num_nodes_n0 = hete.num_nodes()['n0']
    num_nodes_n1 = hete.num_nodes()['n1']

#     num_nodes_n0 = len(hete.node_type['n0'])
#     num_nodes_n1 = len(hete.node_type['n1'])

  #########################################

    return num_nodes_n0, num_nodes_n1

if 'IS_GRADESCOPE_ENV' not in os.environ:
  num_nodes_n0, num_nodes_n1 = get_nodes_per_type(hete)
  print("Node type n0 has {} nodes".format(num_nodes_n0))
  print("Node type n1 has {} nodes".format(num_nodes_n1))
```

```
Node type n0 has 17 nodes
Node type n1 has 17 nodes
```

Question 1.4: Message Types - How many edges are of each message type (10 Points)

When working with heterogenous graphs, as we have discussed before, we now work with heterogenous message types (i.e. different message types for the different `node_type` and `edge_type` combinations). For example, an edge of type `e0` connecting two nodes in club "Mr. HI" would have a message type of (`n0` , `e0` , `n0`). In this problem we will analyze how many edges in our graph are of each message type.

Hint: If you want to learn more about what the different message types are try the call `hete.message_types`

In [18]:
```
print("node_types:{}".format(hete.node_types))
print("Types of messages: \n",hete.message_types)
```

```
node_types:['n0', 'n1']
Types of messages:
 [('n0', 'e0', 'n0'), ('n0', 'e2', 'n1'), ('n1', 'e1', 'n1')]
```

In [19]:
```
print(hete.num_edges())
hete.num_edges(('n0', 'e0', 'n0'))
```

```
{('n0', 'e0', 'n0'): 35, ('n0', 'e2', 'n1'): 11, ('n1', 'e1', 'n1'): 32}
```

Out[19]:
```
35
```

In [20]:
```
def get_num_message_edges(hete):
  # TODO: Implement this function that takes a DeepSNAP dataset object
  # and return the number of edges for each message type.
  # You should return a list of tuples as
  # (message_type, num_edge)

    message_type_edges = []

  ############# Your code here ############
  ## (~2 line of code)
  ## Note
  ## 1. Colab autocomplete functionality might be useful.

    for msg in hete.message_types:
        message_type_edges.append((msg, hete.num_edges(msg)))

  #########################################

    return message_type_edges

if 'IS_GRADESCOPE_ENV' not in os.environ:
  message_type_edges = get_num_message_edges(hete)
  for (message_type, num_edges) in message_type_edges:
    print("Message type {} has {} edges".format(message_type, num_edges))
```

```
Message type ('n0', 'e0', 'n0') has 35 edges
Message type ('n0', 'e2', 'n1') has 11 edges
Message type ('n1', 'e1', 'n1') has 32 edges
```

Question 1.5: Dataset Splitting - How many nodes are in each dataset split? (10 Points)

DeepSNAP has built in Dataset creation and splitting methods for heterogeneous graphs. Here we will create train, validation, and test datasets for a node prediction task and inspect the resulting subgraphs. Specifically, write a function that computes the number of nodes with a known label in each dataset split.

In [21]:
```
for node_type in hete.node_label_index:
    num_nodes = int(len(hete.node_label_index[node_type]))
    print("Number of nodes for node_type {}:{}".format(node_type,num_nodes))
```

```
Number of nodes for node_type n0:17
Number of nodes for node_type n1:17
```

In [22]:
```
from deepsnap.dataset import GraphDataset
```

```
def compute_dataset_split_counts(datasets):
  # TODO: Implement a function that takes a dict of datasets in the form
  # {'train': dataset_train, 'val': dataset_val, 'test': dataset_test}
  # and returns a dict mapping dataset names to the number of labeled
  # nodes used for supervision in that respective dataset.

    data_set_splits = {}

  ############# Your code here ############
  ## (~3 line of code)
  ## Note
  ## 1. The DeepSNAP `node_label_index` dictionary will be helpful.
  ## 2. Remember to count both node_types
  ## 3. Remember each dataset only has one graph that we need to access
  ##    (i.e. dataset[0])
    for item, dataset in datasets.items():
#         print(item,dataset)
#         dataset=datasets[item]
#         print(dataset)

        num_nodes=0
#         num_0, num_1 = get_nodes_per_type(dataset)
#         num_nodes = num_0 + num_1

        for node_type in dataset[0].node_label_index:
#             print(node_type)
            num_nodes = num_nodes+int(len(dataset[0].node_label_index[node_ty
#             print(dataset[node_type])
            #num_nodes = num_nodes + int(len(item[node_type]))

        data_set_splits.update({item:num_nodes})



  #########################################

    return data_set_splits

if 'IS_GRADESCOPE_ENV' not in os.environ:
  dataset = GraphDataset([hete], task='node')
  # Splitting the dataset
  dataset_train, dataset_val, dataset_test = dataset.split(transductive=True,
  datasets = {'train': dataset_train, 'val': dataset_val, 'test': dataset_tes

  data_set_splits = compute_dataset_split_counts(datasets)
  for dataset_name, num_nodes in data_set_splits.items():
    print("{} dataset has {} nodes".format(dataset_name, num_nodes))
```

```
train dataset has 12 nodes
val dataset has 10 nodes
test dataset has 12 nodes
```

DeepSNAP Dataset Visualization

We can now visualize the different nodes and edges used in each graph dataset split.

In [23]:

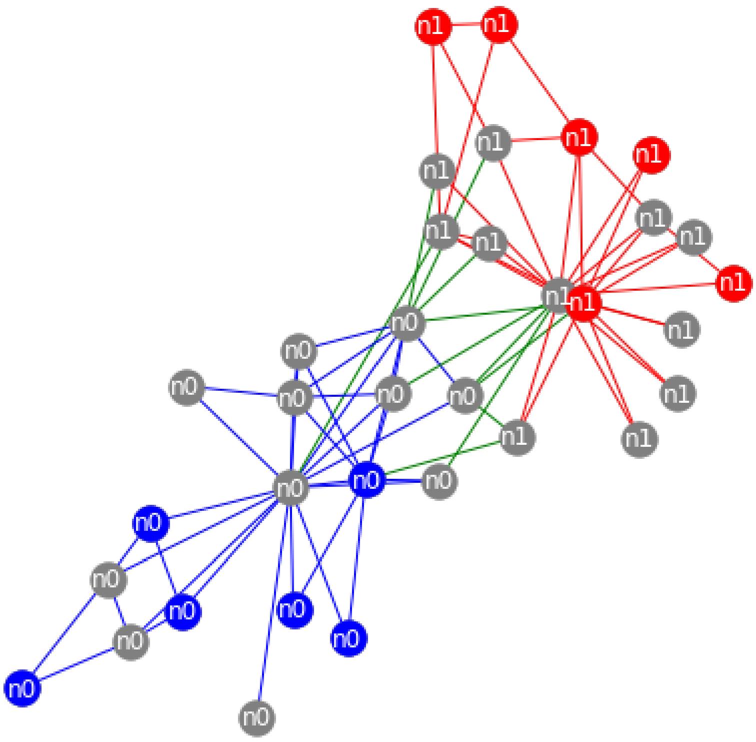
```
dataset_train[0].node_label_index
```

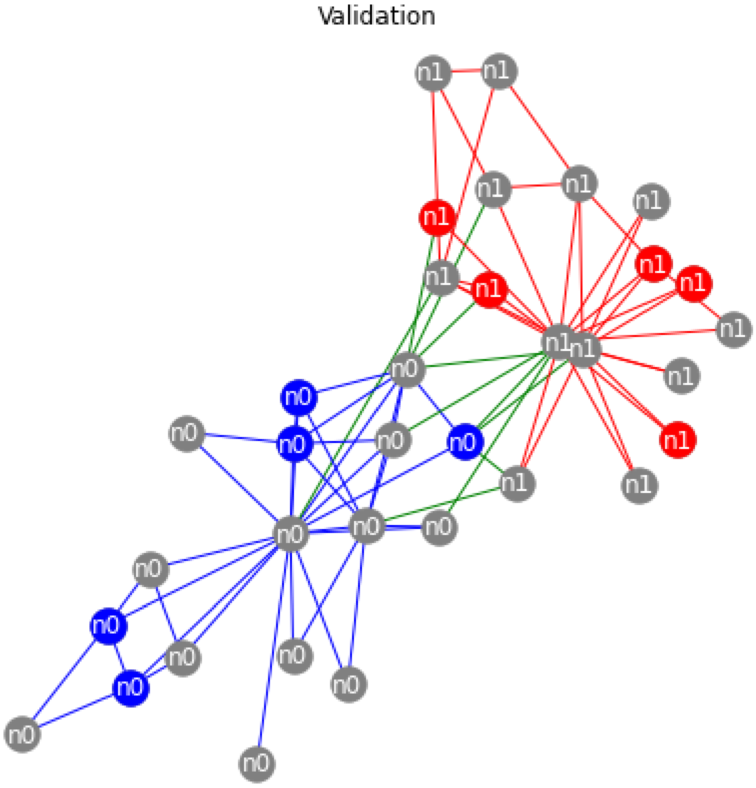
Out[23]:

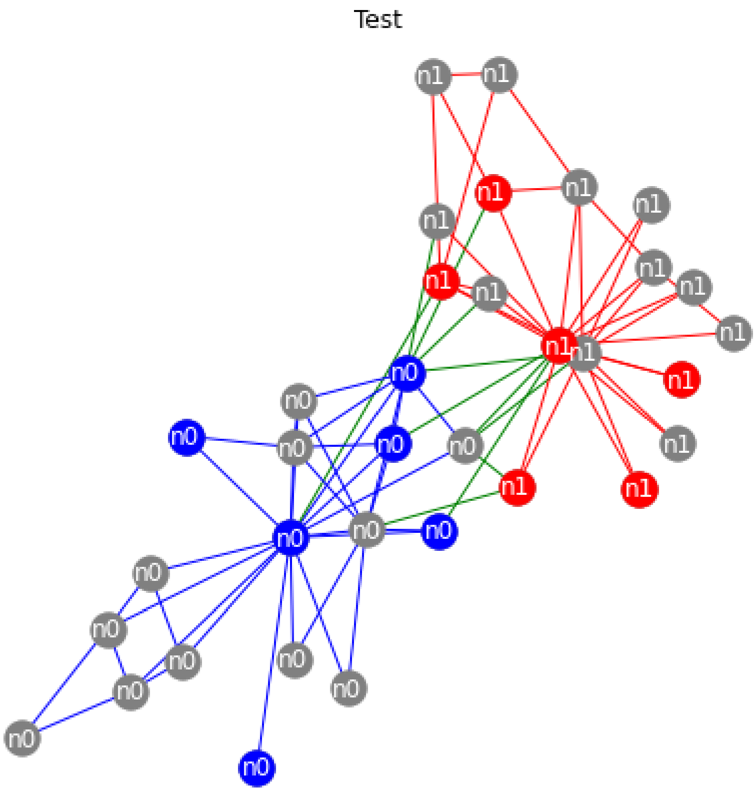
```
{'n0': tensor([15,  2,  0, 10,  1,  7]),
 'n1': tensor([10,  3,  8,  7,  5,  2])}
```

In [24]:
```
from deepsnap.dataset import GraphDataset

if 'IS_GRADESCOPE_ENV' not in os.environ:
  dataset = GraphDataset([hete], task='node')
  # Splitting the dataset
  dataset_train, dataset_val, dataset_test = dataset.split(transductive=True,
  titles = ['Train', 'Validation', 'Test']

  for i, dataset in enumerate([dataset_train, dataset_val, dataset_test]):
    n0 = hete._convert_to_graph_index(dataset[0].node_label_index['n0'], 'n0'
    n1 = hete._convert_to_graph_index(dataset[0].node_label_index['n1'], 'n1'
#    n0 = hete._convert_to_graph_index(dataset[0].node_label_index[0], 'n0').
#    n1 = hete._convert_to_graph_index(dataset[0].node_label_index[1], 'n1').

    plt.figure(figsize=(7, 7))
    plt.title(titles[i])
    nx.draw(G_orig, pos=pos, node_color="grey", edge_color=colors, labels=lab
    nx.draw_networkx_nodes(G_orig.subgraph(n0), pos=pos, node_color="blue")
    nx.draw_networkx_nodes(G_orig.subgraph(n1), pos=pos, node_color="red")
    show()
```

Train







2) Heterogeneous Graph Node Property Prediction

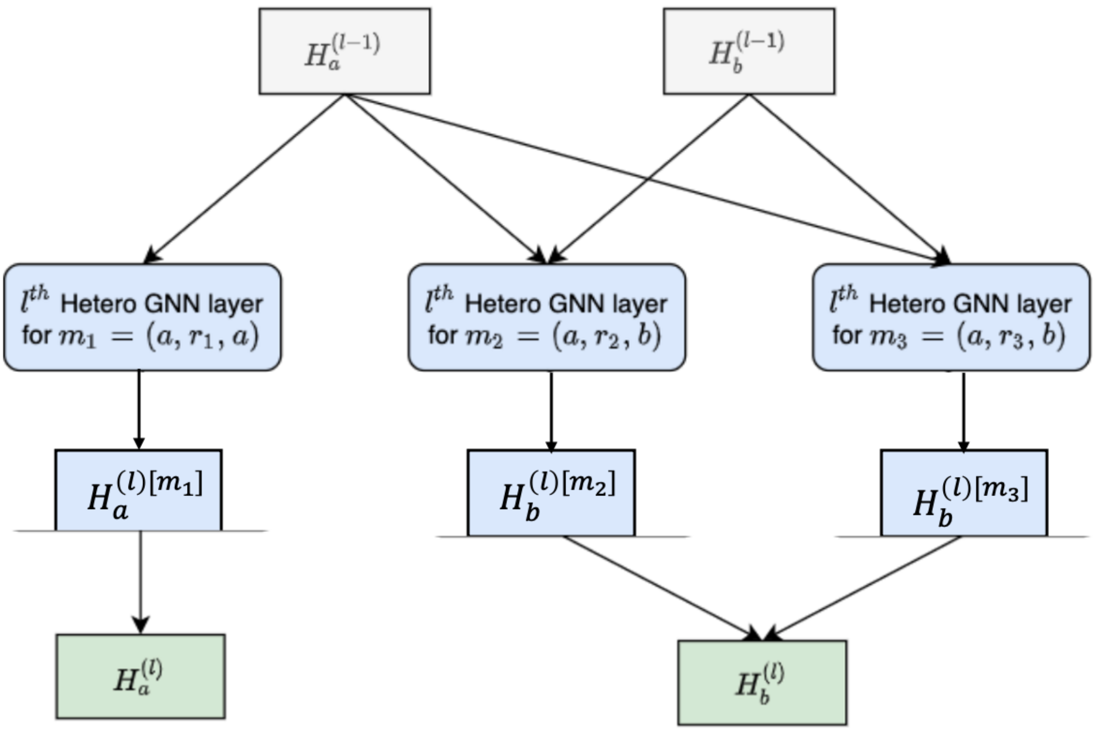
Now, we will use PyTorch Geometric and DeepSNAP to implement a GNN model for heterogeneous graph node property prediction (node classification). We will draw upon our understanding of heterogeneous graphs from lecture and previous work in implementing GNN layers using PyG (introduced in Colab 3).

First let's take a look at the general structure of a heterogeneous GNN layer by working through an example:

Let's assume we have a graph G, which contains two node types a and b, and three message types $m_1 = (a, r_1, a)$, $m_2 = (a, r_2, b)$ and $m_3 = (a, r_3, b)$. Note: during message passing we view each message as (src, relation, dst), where messages "flow" from src to dst node types. For example, during message passing, updating node type b relies on two different message types m_2 and m_3.

When applying message passing in heterogenous graphs, we seperately apply message passing over each message type. Therefore, for the graph G, a heterogeneous GNN layer contains three seperate Heterogeneous Message Passing layers (`HeteroGNNConv` in this Colab), where each `HeteroGNNConv` layer performs message passing and aggregation with respect to *only one message type*. Since a message type is viewed as (src, relation, dst) and messages "flow" from src to dst, each `HeteroGNNConv` layer only computes embeddings for the *dst* nodes of a given message type. For example, the `HeteroGNNConv` layer for message type m_2 outputs updated embedding representations *only* for node's with type b.

An overview of the heterogeneous layer we will create is shown below:



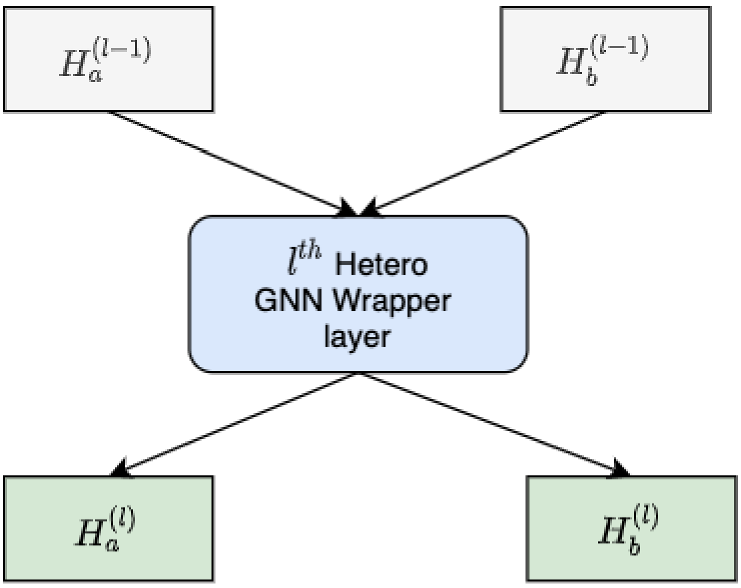
where we highlight the following notation:

- $H_a^{(l)[m_1]}$ is the intermediate matrix of of node embeddings for node type a, generated by the lth `HeteroGNNConv` layer for message type m_1.
- $H_a^{(l)}$ is the matrix with current embeddings for nodes of type a after the lth layer of our Heterogeneous GNN model. Note that these embeddings can rely on one or more intermediate `HeteroGNNConv` layer embeddings(i.e. $H_b^{(l)}$ combines $H_b^{(l)[m_2]}$ and $H_b^{(l)[m_3]}$).

Since each `HeteroGNNConv` is only applied over a single message type, we additionally define a Heterogeneous GNN Wrapper layer (`HeteroGNNWrapperConv`). This wrapper manages and combines the output of each `HeteroGNNConv` layer in order to generate the complete updated node embeddings for each node type in layer l of our model. More specifically, the l^{th} `HeteroGNNWrapperConv` layer takes as input the node embeddings computed for each message type and node type (e.g. $H_b^{(l)[m_2]}$ and $H_b^{(l)[m_3]}$) and aggregates across message types with the same dst node type. The resulting output of the l^{th} `HeteroGNNWrapperConv` layer is the updated embedding matrix $H_i^{(l)}$ for each node type i.

Continuing on our example above, to compute the node embeddings $H_b^{(l)}$ the wrapper layer aggregates output embeddings from the `HeteroGNNConv` layers associated with message types m_2 and m_3 (i.e. $H_b^{(l)[m_2]}$ and $H_b^{(l)[m_3]}$).

With the `HeteroGNNWrapperConv` module, we can now draw a "simplified" heterogeneous layer structure as follows:



NOTE: As reference, it may be helpful to additionally read through PyG's introduciton to heterogeneous graph representations and buidling heterogeneous GNN models: https://pytorch-geometric.readthedocs.io/en/latest/notes/heterogeneous.html

Looking ahead, we recommend you implement the heterogeneous GNN model in following steps:

1. Implement `HeteroGNNConv` .
2. Implement **just** `mean` aggregation within `HeteroGNNWrapperConv` .
3. Implement `generate_convs` .
4. Implement the `HeteroGNN` model and the `train` function.
5. Train the model with `mean` aggregation and test your model to make sure your model has reasonable performance.

6. Once you are confident in your mean aggregation model, implement `attn` aggregation in `HeteroGNNWrapperConv` .
7. Train the model with `attn` aggregation and test your model to make sure your model has reasonable performance.

Note: The key point of advice is to work completely through implementing the mean aggregation heterogeneous GNN model before diving into the more difficult attention based model.

Setup

In [25]:
```
!python -V
```

```
/bin/bash: /home/arch/anaconda3/envs/tf1.15_py3.8_gpu/lib/libtinfo.so.6: no v
ersion information available (required by /bin/bash)
Python 3.8.8
```

In [26]:
```
import copy
import torch
import deepsnap
import numpy as np
import torch.nn as nn
import torch.nn.functional as F
import torch_geometric.nn as pyg_nn

from sklearn.metrics import f1_score
from deepsnap.hetero_gnn import forward_op
from deepsnap.hetero_graph import HeteroGraph
from torch_sparse import SparseTensor, matmul
```

Dataset

You need to login to your Google account and enter the verification code below.

This section has been circumvented by running this code section on google.colab directly, obtaining the acm.pkl file, then downloading it on my hard drive directly.

In [110…
```
# if 'IS_GRADESCOPE_ENV' not in os.environ:
#   from pydrive.auth import GoogleAuth
#   from pydrive.drive import GoogleDrive
#   from google.colab import auth
#   from oauth2client.client import GoogleCredentials

#   # Authenticate and create the PyDrive client
#   auth.authenticate_user()
#   gauth = GoogleAuth()
#   gauth.credentials = GoogleCredentials.get_application_default()
#   drive = GoogleDrive(gauth)
```

In [ ]:
```
# from pydrive.auth import GoogleAuth
# from pydrive.drive import GoogleDrive
# from google.colab import auth
# from oauth2client.client import GoogleCredentials

#   # Authenticate and create the PyDrive client
# auth.authenticate_user()
# gauth = GoogleAuth()
```

```
# gauth.credentials = GoogleCredentials.get_application_default()
# drive = GoogleDrive(gauth)
```

In [ ]:
```
# if 'IS_GRADESCOPE_ENV' not in os.environ:
#   id='1ivlxd6lJMcZ9taS44TMGG72x2V1GeVvk'
#   downloaded = drive.CreateFile({'id': id})
#   downloaded.GetContentFile('acm.pkl')
```

Implementing `HeteroGNNConv`

Now let's start working on our own implementation of the heterogeneous message passing layer (`HeteroGNNConv`)! Just as in Colabs 3 and 4, we will implement the layer using PyTorch Geometric.

At a high level, the `HeteroGNNConv` layer is equivalent to the homogenous GNN layers we implemented in Colab 3, but now applied to an individual heterogeous message type. Moreover, our heterogeneous GNN layer draws directly from the **GraphSAGE** message passing model (Hamilton et al. (2017)).

We begin by defining the `HeteroGNNConv` layer with respect to message type m:

$$m = (s, r, d) \tag{1}$$

where each message type is a tuple containing three elements: s - the source node type, r - the edge (relation) type, and d - the destination node type.

The message passing update rule that we implement is very similar to that of GraphSAGE, except we now need to include the node types and the edge relation type. The update rule for message type m is described below:

$$h_v^{(l)[m]} = W^{(l)[m]} \cdot \text{CONCAT}\Big(W_d^{(l)[m]} \cdot h_v^{(l-1)}, W_s^{(l)[m]} \cdot AGG(\{h_u^{(l-1)}, \forall u \in N_m(v)\})\Big)$$

where we compute $h_v^{(l)[m]}$, the node embedding representation for node v after `HeteroGNNConv` layer l with respect message type m. Further unpacking the forumla we have:

- $W_s^{(l)[m]}$ - linear transformation matrix for the messages of neighboring source nodes of type s along message type m.
- $W_d^{(l)[m]}$ - linear transformation matrix for the message from the node v itself of type d.
- $W^{(l)[m]}$ - linear transformation matrix for the concatenated messages from neighboring node's and the central node.
- $h_u^{(l-1)}$ - the hidden embedding representation for node u after the $(l-1)$th `HeteroGNNWrapperConv` layer. Note, that this embedding is not associated with a particular message type (see layer diagrams above).
- $N_m(v)$ - the set of neighbor source nodes s for the node v that we are embedding along message type $m = (s, r, d)$.

NOTE: We emphasize that each weight matrix is associated with a specific message type $[m]$ and additionally, the weight matrices applied to node messages are differentiated by node type (i.e. W_s and W_d).

Lastly, for simplicity, we use mean aggregations for AGG where:

$$AGG(\{h_u^{(l-1)}, \forall u \in N_m(v)\}) = \frac{1}{|N_m(v)|} \sum_{u \in N_m(v)} h_u^{(l-1)} \tag{3}$$

In [27]:

```
# class HeteroGNNConv(pyg_nn.MessagePassing):
#     def __init__(self, in_channels_src, in_channels_dst, out_channels):
#         super(HeteroGNNConv, self).__init__(aggr="mean")

#         self.in_channels_src = in_channels_src
#         self.in_channels_dst = in_channels_dst
#         self.out_channels = out_channels

#         # To simplify implementation, please initialize both self.lin_dst
#         # and self.lin_src out_features to out_channels
#         self.lin_dst = None
#         self.lin_src = None

#         self.lin_update = None

#         ############# Your code here #############
#         ## (~3 lines of code)
#         ## Note:
#         ## 1. Initialize the 3 linear layers.
#         ## 2. Think through the connection between the mathematical
#         ##    definition of the update rule and torch linear layers!
#         self.lin_dst = nn.Linear(self.in_channels_dst, self.out_channels)
#         self.lin_src = nn.Linear(self.in_channels_src, self.out_channels)
#         self.lin_update = nn.Linear(self.out_channels*2, self.out_channels)


#         ##########################################

#     def forward(
#         self,
#         node_feature_src,
#         node_feature_dst,
#         edge_index,
#         size=None
#     ):
#         ############# Your code here #############
#         ## (~1 line of code)
#         ## Note:
#         ## 1. Unlike Colabs 3 and 4, we just need to call self.propagate wi
#         ## proper/custom arguments.

# #         return self.propagate(edge_index, size=size, node_feature_src=node
# #                              node_feature_dst=node_feature_dst, res_n_id=
#         return self.propagate(edge_index, size=size, node_feature_src=node_
#                              node_feature_dst=node_feature_dst)

#         ##########################################

#     def message_and_aggregate(self, edge_index, node_feature_src):

#         out = None
#         ############# Your code here #############
#         ## (~1 line of code)
```

```
#         ## Note:
#         ## 1. Different from what we implemented in Colabs 3 and 4, we use
#         ##    to combine the previously seperate message and aggregate func
#         ##    The benefit is that we can avoid materializing x_i and x_j
#         ##    to make the implementation more efficient.
#         ## 2. To implement efficiently, refer to PyG documentation for mess
#         ##    and sparse-matrix multiplication:
#         ##    https://pytorch-geometric.readthedocs.io/en/latest/notes/spar
#         ## 3. Here edge_index is torch_sparse SparseTensor. Although intere
#         ##    do not need to deeply understand SparseTensor represenations!
#         ## 4. Conceptually, think through how the message passing and aggre
#         ##    expressed mathematically can be expressed through matrix mult

#         import torch_scatter
#         from torch_sparse import matmul

#         out = matmul(edge_index, node_feature_src, reduce="mean")

#         ##########################################

#         return out

#     def update(self, aggr_out, node_feature_dst):

#         ############# Your code here #############
#         ## (~4 lines of code)
#         ## Note:
#         ## 1. The update function is called after message_and_aggregate
#         ## 2. Think through the one-one connection between the mathematical
#         ##    rule and the 3 linear layers defined in the constructor.

#         aggr_out = self.lin_src(aggr_out)
#         node_feature_dst = self.lin_dst(node_feature_dst)

#         agg_features = torch.cat((node_feature_dst, aggr_out), dim=-1)
#         aggr_out = self.lin_update(agg_features)


#         ##########################################

#         return aggr_out
```

In [67]:

```
### from sample Colab5

class HeteroGNNConv(pyg_nn.MessagePassing):
    def __init__(self, in_channels_src, in_channels_dst, out_channels):
        super(HeteroGNNConv, self).__init__(aggr="mean")

        self.in_channels_src = in_channels_src
        self.in_channels_dst = in_channels_dst
        self.out_channels = out_channels

        # To simplify implementation, please initialize both self.lin_dst
        # and self.lin_src out_features to out_channels
        self.lin_dst = None
        self.lin_src = None

        self.lin_update = None

        ############# Your code here #############
        ## (~3 lines of code)
        self.lin_dst = nn.Linear(self.in_channels_dst, self.out_channels)
        self.lin_src = nn.Linear(self.in_channels_src, self.out_channels)
```

```
        self.lin_update = nn.Linear(self.out_channels*2, self.out_channels)
        ##########################################

    def forward(
        self,
        node_feature_src,
        node_feature_dst,
        edge_index,
        size=None,
        res_n_id=None,
    ):
        ############# Your code here #############
        ## (~1 line of code)
        return self.propagate(edge_index, size=size,
        node_feature_src=node_feature_src,
        node_feature_dst=node_feature_dst,
        res_n_id=res_n_id)

        ##########################################

    def message_and_aggregate(self, edge_index, node_feature_src):

        ############# Your code here #############
        ## (~1 line of code)
        ## Note:
        ## 1. Different from what we implemented in Colab 3, we use message_a
        ## to replace the message and aggregate. The benefit is that we can a
        ## materializing x_i and x_j, and make the implementation more effici
        ## 2. To implement efficiently, following PyG documentation is helpfu
        ## https://pytorch-geometric.readthedocs.io/en/latest/notes/sparse_te
        ## 3. Here edge_index is torch_sparse SparseTensor.

        out = matmul(edge_index, node_feature_src, reduce="mean")
        ##########################################

        return out

    def update(self, aggr_out, node_feature_dst, res_n_id):

        ############# Your code here #############
        ## (~4 lines of code)
        aggr_out = self.lin_src(aggr_out)
        node_feature_dst = self.lin_dst(node_feature_dst)

        concat_features = torch.cat((node_feature_dst, aggr_out), dim=-1)
        aggr_out = self.lin_update(concat_features)
        ##########################################

        return aggr_out
```

Heterogeneous GNN Wrapper Layer

After implementing the `HeteroGNNConv` layer for each message type, we need to manage and aggregate the node embedding results (with respect to each message types). Here we will implement two types of message type level aggregation.

The first one is simply mean aggregation over message types:

$$h_v^{(l)} = \frac{1}{M} \sum_{m=1}^{M} h_v^{(l)[m]} \tag{4}$$

where node v has node type d and we sum over the M message types that have destination node type d. From our original example, for a node v of type b we aggregate v's `HeteroGNNConv` embeddings for message types m_2 and m_3 (i.e. $h_v^{(l)[m_2]}$ and $h_v^{(l)[m_3]}$).

The second method we implement is the semantic level attention introduced in **HAN** ([Wang et al. (2019)](https://arxiv.org/abs/1903.07293)). Instead of directly averaging on the message type aggregation results, we use attention to learn which message type result is more important, then aggregate across all the message types. Below are the equations for semantic level attention:

$$e_m = \frac{1}{|V_d|} \sum_{v \in V_d} q_{attn}^T \cdot tanh\Big(W_{attn}^{(l)} \cdot h_v^{(l)[m]} + b\Big) \tag{5}$$

where m is the message type and d refers to the destination node type for that message ($m = (s, r, d)$). Additionally, V_d refers to the set of nodes v with type d. Lastly, the unormalized attention weight e_m is a scaler computed for each message type m.

Next, we can compute the normalized attention weights and update $h_v^{(l)}$:

$$\alpha_m = \frac{\exp(e_m)}{\sum_{m=1}^M \exp(e_m)} \tag{6}$$

$$h_v^{(l)} = \sum_{m=1}^{M} \alpha_m \cdot h_v^{(l)[m]} \tag{7}$$

, where we emphasize that M here is the number of message types associated with the destination node type d.

Note: The implementation of the attention aggregation is tricky and nuanced. We strongly recommend working carefully through the math equations to undersatnd exactly what each notation refers to and how all the pieces fit together. If you can, try to connect the math to our original example, focusing on node type b, which depends on two different message types!

_We've implemented most of this for you but you'll need to initialize self.attn*proj in the initializer*

For Debugging

Error is in HeteroGNNWrapperConv!!

In [75]:
```
class HeteroGNNWrapperConv(deepsnap.hetero_gnn.HeteroConv):
    def __init__(self, convs, args, aggr="mean"):
        super(HeteroGNNWrapperConv, self).__init__(convs, None)
        self.aggr = aggr

        # Map the index and message type
        self.mapping = {}

        # A numpy array that stores the final attention probability
        self.alpha = None

        self.attn_proj = None

        if self.aggr == "attn":
```

```
            ############# Your code here #############
            ## (~1 line of code)
            ## Note:
            ## 1. Initialize self.attn_proj, where self.attn_proj should incl
            ##    two linear layers. Note, make sure you understand
            ##    which part of the equation self.attn_proj captures.
            ## 2. You should use nn.Sequential for self.attn_proj
            ## 3. nn.Linear and nn.Tanh are useful.
            ## 4. You can model a weight vector (rather than matrix) by using
            ##    nn.Linear(some_size, 1, bias=False).
            ## 5. The first linear layer should have out_features as args['at
            ## 6. You can assume we only have one "head" for the attention.
            ## 7. We recommend you to implement the mean aggregation first. A
            ##    the mean aggregation works well in the training, then you c
            ##    implement this part.

            self.attn_proj = nn.Sequential(
                nn.Linear(args['hidden_size'], args['attn_size']),
                nn.Tanh(),
                nn.Linear(args['attn_size'], 1, bias=False),
            )

            ##########################################

    def reset_parameters(self):
        super(HeteroConvWrapper, self).reset_parameters()
        if self.aggr == "attn":
            for layer in self.attn_proj.children():
                layer.reset_parameters()

    def forward(self, node_features, edge_indices):
        message_type_emb = {}
        for message_key, message_type in edge_indices.items():
            src_type, edge_type, dst_type = message_key
            node_feature_src = node_features[src_type]
            node_feature_dst = node_features[dst_type]
            edge_index = edge_indices[message_key]
            message_type_emb[message_key] = (
                self.convs[message_key](
                    node_feature_src,
                    node_feature_dst,
                    edge_index,
                )
            )
        node_emb = {dst: [] for _, _, dst in message_type_emb.keys()}
        mapping = {}
        for (src, edge_type, dst), item in message_type_emb.items():
            mapping[len(node_emb[dst])] = (src, edge_type, dst)
            node_emb[dst].append(item)
        self.mapping = mapping
        for node_type, embs in node_emb.items():
            if len(embs) == 1:
                node_emb[node_type] = embs[0]
            else:
                node_emb[node_type] = self.aggregate(embs)
        return node_emb

    def aggregate(self, xs):
        # TODO: Implement this function that aggregates all message type resu
        # Here, xs is a list of tensors (embeddings) with respect to message
        # type aggregation results.

        if self.aggr == "mean":
```

```
            ############# Your code here #############
            ## (~2 lines of code)
            ## Note:
            ## 1. Explore the function parameter `xs`!

            out = torch.mean(torch.stack(xs), dim=0)
            return out

            ##########################################

            ############ AC Comment: There seems to be an error with this coc
        elif self.aggr == "attn":
            N = xs[0].shape[0] # Number of nodes for that node type
            M = len(xs) # Number of message types for that node type

            x = torch.cat(xs, dim=0).view(M, N, -1) # M * N * D
            z = self.attn_proj(x).view(M, N) # M * N * 1
            z = z.mean(1) # M * 1
            alpha = torch.softmax(z, dim=0) # M * 1

            # Store the attention result to self.alpha as np array
            self.alpha = alpha.view(-1).data.cpu().numpy()

            alpha = alpha.view(M, 1, 1)
            x = x * alpha
            return x.sum(dim=0)

#             ## code source: https://notebooks.githubusercontent.com/view/ip
#             x = self.attn_proj(torch.stack(xs, dim=0))
#             x = torch.mean(x, dim=1)

#             self.alpha = torch.softmax(x, dim=0)
#             self.alpha = self.alpha.detach()

#             out = torch.stack(xs, dim=0)
#             out = self.alpha.unsqueeze(-1) * out

#             out = torch.sum(out, dim=0)
#             return out
```

Initialize Heterogeneous GNN Layers

Now let's put it all together and initialize the Heterogeneous GNN Layers. Different from the homogeneous graph case, heterogeneous graphs can be a little bit complex.

In general, we need to create a dictionary of `HeteroGNNConv` layers where the keys are message types.

- To get all message types, `deepsnap.hetero_graph.HeteroGraph.message_types` is useful.
- If we are initializing the first conv layers, we need to get the feature dimension of each node type. Using `deepsnap.hetero_graph.HeteroGraph.num_node_features(node_type)` will return the node feature dimension of `node_type`. In this function, we will set each `HeteroGNNConv` `out_channels` to be `hidden_size`.
- If we are not initializing the first conv layers, all node types will have the smae embedding dimension `hidden_size` and we still set `HeteroGNNConv` `out_channels` to be

`hidden_size` for simplicity.

In [69]:

```
def generate_convs(hetero_graph, conv, hidden_size, first_layer=False):
    # TODO: Implement this function that returns a dictionary of `HeteroGNNCo
    # layers where the keys are message types. `hetero_graph` is deepsnap `He
    # object and the `conv` is the `HeteroGNNConv`.

    convs = {}

    ############# Your code here #############
    ## (~9 lines of code)
    ## Note:
    ## 1. See the hints above!
    ## 2. conv is of type `HeteroGNNConv`

    for msg_type in hetero_graph.message_types:
        if first_layer:
            num_node_feature_src = hetero_graph.num_node_features(msg_type[0]
            num_node_feature_dst = hetero_graph.num_node_features(msg_type[-1

            convs[msg_type] = conv(num_node_feature_src, num_node_feature_dst

        else:
            convs[msg_type] = conv(hidden_size, hidden_size, hidden_size)


    ##########################################

    return convs
```

HeteroGNN

Now we will make a simple HeteroGNN model which contains only two `HeteroGNNWrapperConv` layers.

For the forward function in `HeteroGNN`, the model is going to be run as following:

$$\text{self.convs1} \rightarrow \text{self.bns1} \rightarrow \text{self.relus1} \rightarrow \text{self.convs2} \rightarrow \text{self.bns2} \rightarrow \text{self.relus2} \rightarrow \text{self.po}$$

In [79]:

```
class HeteroGNN(torch.nn.Module):
    def __init__(self, hetero_graph, args, aggr="mean"):
        super(HeteroGNN, self).__init__()

        self.aggr = aggr
        self.hidden_size = args['hidden_size']

        self.convs1 = None
        self.convs2 = None

        self.bns1 = nn.ModuleDict()
        self.bns2 = nn.ModuleDict()
        self.relus1 = nn.ModuleDict()
        self.relus2 = nn.ModuleDict()
        self.post_mps = nn.ModuleDict()

        ############# Your code here #############
        ## (~10 lines of code)
        ## Note:
```

```
        ## 1. For self.convs1 and self.convs2, call generate_convs at first a
        ##    pass the returned dictionary of `HeteroGNNConv` to `HeteroGNNWr
        ## 2. For self.bns, self.relus and self.post_mps, the keys are node_t
        ##    `deepsnap.hetero_graph.HeteroGraph.node_types` will be helpful.
        ## 3. Initialize all batchnorms to torch.nn.BatchNorm1d(hidden_size,
        ## 4. Initialize all relus to nn.LeakyReLU().
        ## 5. For self.post_mps, each value in the ModuleDict is a linear lay
        ##    where the `out_features` is the number of classes for that node
        ##    `deepsnap.hetero_graph.HeteroGraph.num_node_labels(node_type)`
        ##    useful.

        self.convs1 = generate_convs(hetero_graph, HeteroGNNConv, self.hidden
        self.convs1 = HeteroGNNWrapperConv(self.convs1, args, self.aggr)
        self.convs2 = generate_convs(hetero_graph, HeteroGNNConv, self.hidden
        self.convs2 = HeteroGNNWrapperConv(self.convs2, args, self.aggr)

        for node_type in hetero_graph.node_types:
            self.bns1[node_type] = torch.nn.BatchNorm1d(self.hidden_size, eps
            self.bns2[node_type] = torch.nn.BatchNorm1d(self.hidden_size, eps
            self.relus1[node_type] = nn.LeakyReLU()
            self.relus2[node_type] = nn.LeakyReLU()
            self.post_mps[node_type] = nn.Linear(self.hidden_size, hetero_gra

        ##########################################

    def forward(self, node_feature, edge_index):
        # TODO: Implement the forward function. Notice that `node_feature` is
        # a dictionary of tensors where keys are node types and values are
        # corresponding feature tensors. The `edge_index` is a dictionary of
        # tensors where keys are message types and values are corresponding
        # edge index tensors (with respect to each message type).

        x = node_feature

        ############# Your code here #############
        ## (~7 lines of code)
        ## Note:
        ## 1. `deepsnap.hetero_gnn.forward_op` can be helpful.

        #print(x)
        #print(edge_index)
        x = self.convs1(x, edge_index)
        x = forward_op(x, self.bns1)
        x = forward_op(x, self.relus1)

        x = self.convs2(x, edge_index)
        x = forward_op(x, self.bns2)
        x = forward_op(x, self.relus2)

        x = forward_op(x, self.post_mps)

        ##########################################

        return x

    def loss(self, preds, y, indices):

        loss = 0
        loss_func = torch.nn.functional.cross_entropy
#           loss_func = F.cross_entropy
        ############# Your code here #############
        ## (~3 lines of code)
        ## Note:
        ## 1. For each node type in preds, accumulate computed loss to `loss`
```

```
        ## 2. Loss need to be computed with respect to the given index
        ## 3. preds is a dictionary of model predictions keyed by node_type.
        ## 4. indeces is a dictionary of labeled supervision nodes keyed
        ##    by node_type

        for node_type in preds:
            idx = indices[node_type]
            pred = preds[node_type][idx]
            loss += loss_func(pred, y[node_type][idx])

        ##########################################

        return loss
```

Start of debugging code

In [76]:

```
### this is for debugging

best_model = None
best_val = 0

output_size = hetero_graph.num_node_labels('paper')
model = HeteroGNN(hetero_graph, args, aggr="attn").to(args['device'])
optimizer = torch.optim.Adam(model.parameters(), lr=args['lr'], weight_decay=

  #for epoch in range(args['epochs']):

#loss = train(model, optimizer, hetero_graph, train_idx)
model.train()
optimizer.zero_grad()
print(hetero_graph.node_feature)
print(hetero_graph.edge_index)
preds = model(hetero_graph.node_feature, hetero_graph.edge_index)
print(preds)
    ############# Your code here ############
    ## Note:
    ## 1. Compute the loss here
    ## 2. `deepsnap.hetero_graph.HeteroGraph.node_label` is useful

#loss = model.loss(preds, hetero_graph.node_label, train_idx)


    ##########################################

#loss.backward()
#optimizer.step()
#print(loss.item())
```

```
{'paper': tensor([[1., 1., 1.,  ..., 0., 0., 0.],
        [0., 1., 0.,  ..., 0., 0., 0.],
        [0., 1., 0.,  ..., 0., 0., 0.],
        ...,
        [1., 1., 0.,  ..., 0., 0., 0.],
        [0., 1., 0.,  ..., 0., 0., 0.],
        [0., 0., 1.,  ..., 0., 0., 0.]])}
{('paper', 'author', 'paper'): SparseTensor(row=tensor([   0,    0,    0,
..., 3024, 3024, 3024]),
             col=tensor([   8,   20,   51,  ..., 2948, 2983, 2991]),
             size=(3025, 3025), nnz=26256, density=0.29%), ('paper', 'subjec
t', 'paper'): SparseTensor(row=tensor([   0,    0,    0,  ..., 3024, 3024, 30
24]),
             col=tensor([  75,  434,  534,  ..., 3020, 3021, 3022]),
             size=(3025, 3025), nnz=2207736, density=24.13%)}
```

```
{'paper': tensor([[ 0.1050, -0.0127, -0.0403],
        [ 0.1046, -0.0098, -0.0387],
        [ 0.1048, -0.0103, -0.0384],
        ...,
        [ 0.1049, -0.0118, -0.0391],
        [ 0.1045, -0.0091, -0.0388],
        [ 0.1056, -0.0105, -0.0404]], grad_fn=<AddmmBackward0>)}
```

In [46]:

```
model
```

Out[46]:

```
HeteroGNN(
  (bns1): ModuleDict(
    (paper): BatchNorm1d(64, eps=1.0, momentum=0.1, affine=True, track_runnin
g_stats=True)
  )
  (bns2): ModuleDict(
    (paper): BatchNorm1d(64, eps=1.0, momentum=0.1, affine=True, track_runnin
g_stats=True)
  )
  (relus1): ModuleDict(
    (paper): LeakyReLU(negative_slope=0.01)
  )
  (relus2): ModuleDict(
    (paper): LeakyReLU(negative_slope=0.01)
  )
  (post_mps): ModuleDict(
    (paper): Linear(in_features=64, out_features=3, bias=True)
  )
  (convs1): HeteroGNNWrapperConv(
    (modules): ModuleList(
      (0): HeteroGNNConv()
      (1): HeteroGNNConv()
    )
    (attn_proj): Sequential(
      (0): Linear(in_features=64, out_features=32, bias=True)
      (1): Tanh()
      (2): Linear(in_features=64, out_features=1, bias=False)
    )
  )
  (convs2): HeteroGNNWrapperConv(
    (modules): ModuleList(
      (0): HeteroGNNConv()
      (1): HeteroGNNConv()
    )
    (attn_proj): Sequential(
      (0): Linear(in_features=64, out_features=32, bias=True)
      (1): Tanh()
      (2): Linear(in_features=64, out_features=1, bias=False)
    )
  )
)
```

In [ ]:

```
def train2(model, optimizer, hetero_graph, train_idx):
    model.train()
    optimizer.zero_grad()
    preds = model(hetero_graph.node_feature, hetero_graph.edge_index)

    loss = None

    ############# Your code here #############
    ## Note:
    ## 1. Compute the loss here
    ## 2. `deepsnap.hetero_graph.HeteroGraph.node_label` is useful

    loss = model.loss(preds, hetero_graph.node_label, train_idx)
```

```
    ##########################################

    loss.backward()
    optimizer.step()
    return loss.item()
```

End of debugging code

Training and Testing

Here we provide you with the functions to train and test. You only need to implement one line of code here.

Please do not modify other parts in `train` and `test` for grading purposes.

In [89]:
```
import pandas as pd

def train(model, optimizer, hetero_graph, train_idx):
    model.train()
    optimizer.zero_grad()
    preds = model(hetero_graph.node_feature, hetero_graph.edge_index)

    loss = None

    ############# Your code here #############
    ## Note:
    ## 1. Compute the loss here
    ## 2. `deepsnap.hetero_graph.HeteroGraph.node_label` is useful

    loss = model.loss(preds, hetero_graph.node_label, train_idx)


    ##########################################

    loss.backward()
    optimizer.step()
    return loss.item()

def test(model, graph, indices, best_model=None, best_val=0, save_preds=False
    model.eval()
    accs = []
    for i, index in enumerate(indices):
        preds = model(graph.node_feature, graph.edge_index)
        num_node_types = 0
        micro = 0
        macro = 0
        for node_type in preds:
            idx = index[node_type]
            pred = preds[node_type][idx]
            pred = pred.max(1)[1]
            label_np = graph.node_label[node_type][idx].cpu().numpy()
            pred_np = pred.cpu().numpy()
            micro = f1_score(label_np, pred_np, average='micro')
            macro = f1_score(label_np, pred_np, average='macro')
            num_node_types += 1

        # Averaging f1 score might not make sense, but in our example we only
        # have one node type
        micro /= num_node_types
```

```
        macro /= num_node_types
        accs.append((micro, macro))

        # Only save the test set predictions and labels!
        if save_preds and i == 2:
##          print ("Saving Heterogeneous Node Prediction Model Predictions wi
          print()

          data = {}
          data['pred'] = pred_np
          data['label'] = label_np

          df = pd.DataFrame(data=data)
          # Save locally as csv
          df.to_csv('ACM-Node-' + agg_type + 'Agg.csv', sep=',', index=False)

    if accs[1][0] > best_val:
        best_val = accs[1][0]
        best_model = copy.deepcopy(model)
    return accs, best_model, best_val
```

In [90]:

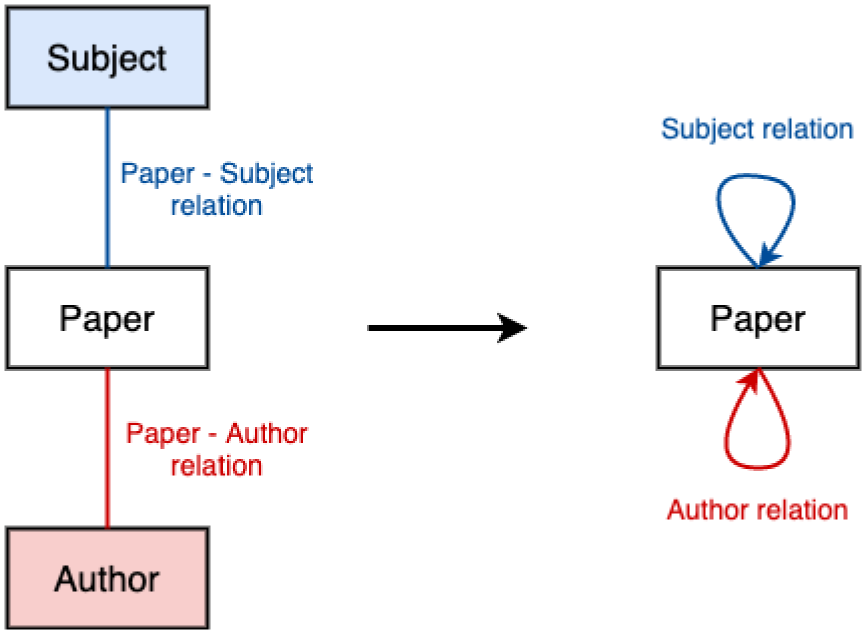
```
# Please do not change the following parameters
args = {
    'device': torch.device('cuda' if torch.cuda.is_available() else 'cpu'),
    'hidden_size': 64,
    'epochs': 100,
    'weight_decay': 1e-5,
    'lr': 0.003,
    'attn_size': 32,
}
```

Dataset and Preprocessing

In the next, we will load the data and create a tensor backend (without a NetworkX graph) `deepsnap.hetero_graph.HeteroGraph` object.

We will use the `ACM(3025)` dataset in our node property prediction task, which is proposed in **HAN** (Wang et al. (2019)) and our dataset is extracted from DGL's ACM.mat.

The original ACM dataset has three node types and two edge (relation) types. For simplicity, we simplify the heterogeneous graph to one node type and two edge types (shown below). This means that in our heterogeneous graph, we have one node type (paper) and two message types (paper, author, paper) and (paper, subject, paper).



In [91]:

```
if 'IS_GRADESCOPE_ENV' not in os.environ:
  print("Device: {}".format(args['device']))

  # Load the data
  data = torch.load("acm.pkl")

  # Message types
  message_type_1 = ("paper", "author", "paper")
  message_type_2 = ("paper", "subject", "paper")

  # Dictionary of edge indices
  edge_index = {}
  edge_index[message_type_1] = data['pap']
  edge_index[message_type_2] = data['psp']

  # Dictionary of node features
  node_feature = {}
  node_feature["paper"] = data['feature']

  # Dictionary of node labels
  node_label = {}
  node_label["paper"] = data['label']

  # Load the train, validation and test indices
  train_idx = {"paper": data['train_idx'].to(args['device'])}
  val_idx = {"paper": data['val_idx'].to(args['device'])}
  test_idx = {"paper": data['test_idx'].to(args['device'])}

  # Construct a deepsnap tensor backend HeteroGraph
  hetero_graph = HeteroGraph(
      node_feature=node_feature,
      node_label=node_label,
      edge_index=edge_index,
      directed=True
  )

  print(f"ACM heterogeneous graph: {hetero_graph.num_nodes()} nodes, {hetero_

  # Node feature and node label to device
  for key in hetero_graph.node_feature:
      hetero_graph.node_feature[key] = hetero_graph.node_feature[key].to(args
  for key in hetero_graph.node_label:
      hetero_graph.node_label[key] = hetero_graph.node_label[key].to(args['de
```

```
    # Edge_index to sparse tensor and to device
    for key in hetero_graph.edge_index:
        edge_index = hetero_graph.edge_index[key]
        adj = SparseTensor(row=edge_index[0], col=edge_index[1], sparse_sizes=(
        hetero_graph.edge_index[key] = adj.t().to(args['device'])
    print(hetero_graph.edge_index[message_type_1])
    print(hetero_graph.edge_index[message_type_2])
```

```
Device: cpu
ACM heterogeneous graph: {'paper': 3025} nodes, {('paper', 'author', 'pape
r'): 26256, ('paper', 'subject', 'paper'): 2207736} edges
SparseTensor(row=tensor([   0,    0,    0,  ..., 3024, 3024, 3024]),
             col=tensor([   8,   20,   51,  ..., 2948, 2983, 2991]),
             size=(3025, 3025), nnz=26256, density=0.29%)
SparseTensor(row=tensor([   0,    0,    0,  ..., 3024, 3024, 3024]),
             col=tensor([  75,  434,  534,  ..., 3020, 3021, 3022]),
             size=(3025, 3025), nnz=2207736, density=24.13%)
```

Start Training!

Now lets start training!

Training the Mean Aggregation

In [83]:

```
if 'IS_GRADESCOPE_ENV' not in os.environ:
    best_model = None
    best_val = 0

    model = HeteroGNN(hetero_graph, args, aggr="mean").to(args['device'])
    optimizer = torch.optim.Adam(model.parameters(), lr=args['lr'], weight_deca

    for epoch in range(args['epochs']):
        loss = train(model, optimizer, hetero_graph, train_idx)
        accs, best_model, best_val = test(model, hetero_graph, [train_idx, val_
        print(
            f"Epoch {epoch + 1}: loss {round(loss, 5)}, "
            f"train micro {round(accs[0][0] * 100, 2)}%, train macro {round(acc
            f"valid micro {round(accs[1][0] * 100, 2)}%, valid macro {round(acc
            f"test micro {round(accs[2][0] * 100, 2)}%, test macro {round(accs[
        )
    best_accs, _, _ = test(best_model, hetero_graph, [train_idx, val_idx, test_
    print(
        f"Best model: "
        f"train micro {round(best_accs[0][0] * 100, 2)}%, train macro {round(be
        f"valid micro {round(best_accs[1][0] * 100, 2)}%, valid macro {round(be
        f"test micro {round(best_accs[2][0] * 100, 2)}%, test macro {round(best
    )
```

```
Epoch 1: loss 1.10229, train micro 33.33%, train macro 16.67%, valid micro 3
3.33%, valid macro 16.67%, test micro 35.81%, test macro 17.58%
Epoch 2: loss 1.09319, train micro 33.33%, train macro 16.67%, valid micro 3
3.33%, valid macro 16.67%, test micro 35.81%, test macro 17.58%
Epoch 3: loss 1.06455, train micro 37.0%, train macro 23.74%, valid micro 33.
33%, valid macro 16.67%, test micro 35.91%, test macro 17.79%
Epoch 4: loss 1.00755, train micro 65.33%, train macro 54.58%, valid micro 6
3.67%, valid macro 53.34%, test micro 56.61%, test macro 47.32%
Epoch 5: loss 0.9104, train micro 67.17%, train macro 56.69%, valid micro 66.
0%, valid macro 54.75%, test micro 64.52%, test macro 53.4%
Epoch 6: loss 0.77037, train micro 67.0%, train macro 59.13%, valid micro 65.
0%, valid macro 53.8%, test micro 62.49%, test macro 50.94%
Epoch 7: loss 0.60802, train micro 67.17%, train macro 61.74%, valid micro 6
```

```
3.33%, valid macro 53.06%, test micro 60.05%, test macro 48.58%
Epoch 8: loss 0.4618, train micro 70.5%, train macro 67.05%, valid micro 65.
0%, valid macro 56.54%, test micro 60.28%, test macro 49.3%
Epoch 9: loss 0.35179, train micro 77.0%, train macro 75.07%, valid micro 71.
0%, valid macro 65.46%, test micro 63.06%, test macro 53.02%
Epoch 10: loss 0.27068, train micro 84.67%, train macro 83.8%, valid micro 7
5.33%, valid macro 71.86%, test micro 66.07%, test macro 57.6%
Epoch 11: loss 0.20834, train micro 91.17%, train macro 90.95%, valid micro 8
1.67%, valid macro 80.41%, test micro 68.28%, test macro 60.96%
Epoch 12: loss 0.15994, train micro 94.5%, train macro 94.44%, valid micro 8
8.0%, valid macro 87.63%, test micro 71.15%, test macro 65.68%
Epoch 13: loss 0.12315, train micro 96.67%, train macro 96.65%, valid micro 9
0.33%, valid macro 90.11%, test micro 74.31%, test macro 70.59%
Epoch 14: loss 0.09588, train micro 98.17%, train macro 98.17%, valid micro 9
2.33%, valid macro 92.25%, test micro 76.94%, test macro 74.48%
Epoch 15: loss 0.07624, train micro 99.33%, train macro 99.33%, valid micro 9
3.67%, valid macro 93.62%, test micro 79.25%, test macro 77.67%
Epoch 16: loss 0.06182, train micro 99.5%, train macro 99.5%, valid micro 95.
33%, valid macro 95.32%, test micro 81.22%, test macro 80.13%
Epoch 17: loss 0.05069, train micro 99.83%, train macro 99.83%, valid micro 9
6.33%, valid macro 96.33%, test micro 83.11%, test macro 82.39%
Epoch 18: loss 0.04171, train micro 99.83%, train macro 99.83%, valid micro 9
7.0%, valid macro 97.01%, test micro 84.89%, test macro 84.43%
Epoch 19: loss 0.03446, train micro 99.83%, train macro 99.83%, valid micro 9
7.0%, valid macro 97.01%, test micro 85.65%, test macro 85.28%
Epoch 20: loss 0.02857, train micro 100.0%, train macro 100.0%, valid micro 9
7.33%, valid macro 97.34%, test micro 86.26%, test macro 85.99%
Epoch 21: loss 0.02379, train micro 100.0%, train macro 100.0%, valid micro 9
7.67%, valid macro 97.67%, test micro 86.49%, test macro 86.26%
Epoch 22: loss 0.01992, train micro 100.0%, train macro 100.0%, valid micro 9
8.0%, valid macro 98.0%, test micro 86.68%, test macro 86.48%
Epoch 23: loss 0.01668, train micro 100.0%, train macro 100.0%, valid micro 9
7.67%, valid macro 97.67%, test micro 86.92%, test macro 86.73%
Epoch 24: loss 0.0139, train micro 100.0%, train macro 100.0%, valid micro 9
7.33%, valid macro 97.34%, test micro 87.06%, test macro 86.87%
Epoch 25: loss 0.01155, train micro 100.0%, train macro 100.0%, valid micro 9
7.33%, valid macro 97.34%, test micro 87.15%, test macro 86.97%
Epoch 26: loss 0.00957, train micro 100.0%, train macro 100.0%, valid micro 9
7.67%, valid macro 97.67%, test micro 87.15%, test macro 86.97%
Epoch 27: loss 0.00796, train micro 100.0%, train macro 100.0%, valid micro 9
7.67%, valid macro 97.67%, test micro 86.96%, test macro 86.79%
Epoch 28: loss 0.00665, train micro 100.0%, train macro 100.0%, valid micro 9
7.67%, valid macro 97.67%, test micro 86.82%, test macro 86.63%
Epoch 29: loss 0.00559, train micro 100.0%, train macro 100.0%, valid micro 9
7.33%, valid macro 97.34%, test micro 86.64%, test macro 86.43%
Epoch 30: loss 0.00475, train micro 100.0%, train macro 100.0%, valid micro 9
7.0%, valid macro 97.01%, test micro 86.35%, test macro 86.13%
Epoch 31: loss 0.00408, train micro 100.0%, train macro 100.0%, valid micro 9
7.0%, valid macro 97.01%, test micro 86.02%, test macro 85.78%
Epoch 32: loss 0.00353, train micro 100.0%, train macro 100.0%, valid micro 9
7.0%, valid macro 97.0%, test micro 85.98%, test macro 85.72%
Epoch 33: loss 0.0031, train micro 100.0%, train macro 100.0%, valid micro 9
6.33%, valid macro 96.33%, test micro 85.93%, test macro 85.67%
Epoch 34: loss 0.00273, train micro 100.0%, train macro 100.0%, valid micro 9
6.33%, valid macro 96.33%, test micro 85.79%, test macro 85.52%
Epoch 35: loss 0.00244, train micro 100.0%, train macro 100.0%, valid micro 9
6.0%, valid macro 96.0%, test micro 85.55%, test macro 85.28%
Epoch 36: loss 0.00219, train micro 100.0%, train macro 100.0%, valid micro 9
5.67%, valid macro 95.67%, test micro 85.46%, test macro 85.19%
Epoch 37: loss 0.00198, train micro 100.0%, train macro 100.0%, valid micro 9
5.67%, valid macro 95.67%, test micro 85.41%, test macro 85.14%
Epoch 38: loss 0.0018, train micro 100.0%, train macro 100.0%, valid micro 9
5.67%, valid macro 95.67%, test micro 85.22%, test macro 84.95%
Epoch 39: loss 0.00165, train micro 100.0%, train macro 100.0%, valid micro 9
5.67%, valid macro 95.67%, test micro 85.22%, test macro 84.95%
Epoch 40: loss 0.00151, train micro 100.0%, train macro 100.0%, valid micro 9
6.0%, valid macro 96.0%, test micro 85.27%, test macro 85.0%
Epoch 41: loss 0.0014, train micro 100.0%, train macro 100.0%, valid micro 9
6.0%, valid macro 96.0%, test micro 85.18%, test macro 84.91%
```

```
Epoch 42: loss 0.0013, train micro 100.0%, train macro 100.0%, valid micro 9
5.67%, valid macro 95.67%, test micro 85.18%, test macro 84.91%
Epoch 43: loss 0.00122, train micro 100.0%, train macro 100.0%, valid micro 9
5.67%, valid macro 95.67%, test micro 85.13%, test macro 84.86%
Epoch 44: loss 0.00114, train micro 100.0%, train macro 100.0%, valid micro 9
5.67%, valid macro 95.67%, test micro 84.8%, test macro 84.54%
Epoch 45: loss 0.00107, train micro 100.0%, train macro 100.0%, valid micro 9
5.67%, valid macro 95.67%, test micro 84.8%, test macro 84.54%
Epoch 46: loss 0.00101, train micro 100.0%, train macro 100.0%, valid micro 9
5.67%, valid macro 95.67%, test micro 84.75%, test macro 84.51%
Epoch 47: loss 0.00096, train micro 100.0%, train macro 100.0%, valid micro 9
5.67%, valid macro 95.67%, test micro 84.71%, test macro 84.47%
Epoch 48: loss 0.00091, train micro 100.0%, train macro 100.0%, valid micro 9
5.67%, valid macro 95.67%, test micro 84.66%, test macro 84.43%
Epoch 49: loss 0.00087, train micro 100.0%, train macro 100.0%, valid micro 9
5.67%, valid macro 95.67%, test micro 84.56%, test macro 84.34%
Epoch 50: loss 0.00083, train micro 100.0%, train macro 100.0%, valid micro 9
5.67%, valid macro 95.67%, test micro 84.56%, test macro 84.35%
Epoch 51: loss 0.0008, train micro 100.0%, train macro 100.0%, valid micro 9
5.67%, valid macro 95.67%, test micro 84.47%, test macro 84.25%
Epoch 52: loss 0.00077, train micro 100.0%, train macro 100.0%, valid micro 9
5.67%, valid macro 95.67%, test micro 84.42%, test macro 84.21%
Epoch 53: loss 0.00074, train micro 100.0%, train macro 100.0%, valid micro 9
5.33%, valid macro 95.34%, test micro 84.56%, test macro 84.36%
Epoch 54: loss 0.00071, train micro 100.0%, train macro 100.0%, valid micro 9
5.33%, valid macro 95.34%, test micro 84.56%, test macro 84.36%
Epoch 55: loss 0.00069, train micro 100.0%, train macro 100.0%, valid micro 9
5.67%, valid macro 95.67%, test micro 84.33%, test macro 84.14%
Epoch 56: loss 0.00067, train micro 100.0%, train macro 100.0%, valid micro 9
5.67%, valid macro 95.67%, test micro 84.33%, test macro 84.15%
Epoch 57: loss 0.00065, train micro 100.0%, train macro 100.0%, valid micro 9
5.67%, valid macro 95.67%, test micro 84.38%, test macro 84.21%
Epoch 58: loss 0.00063, train micro 100.0%, train macro 100.0%, valid micro 9
5.67%, valid macro 95.67%, test micro 84.28%, test macro 84.12%
Epoch 59: loss 0.00061, train micro 100.0%, train macro 100.0%, valid micro 9
5.67%, valid macro 95.67%, test micro 84.19%, test macro 84.03%
Epoch 60: loss 0.0006, train micro 100.0%, train macro 100.0%, valid micro 9
5.67%, valid macro 95.67%, test micro 84.19%, test macro 84.03%
Epoch 61: loss 0.00058, train micro 100.0%, train macro 100.0%, valid micro 9
5.67%, valid macro 95.67%, test micro 84.24%, test macro 84.08%
Epoch 62: loss 0.00057, train micro 100.0%, train macro 100.0%, valid micro 9
5.67%, valid macro 95.67%, test micro 84.24%, test macro 84.09%
Epoch 63: loss 0.00055, train micro 100.0%, train macro 100.0%, valid micro 9
5.67%, valid macro 95.67%, test micro 84.19%, test macro 84.04%
Epoch 64: loss 0.00054, train micro 100.0%, train macro 100.0%, valid micro 9
5.67%, valid macro 95.67%, test micro 84.05%, test macro 83.9%
Epoch 65: loss 0.00053, train micro 100.0%, train macro 100.0%, valid micro 9
5.67%, valid macro 95.67%, test micro 84.09%, test macro 83.96%
Epoch 66: loss 0.00052, train micro 100.0%, train macro 100.0%, valid micro 9
5.67%, valid macro 95.67%, test micro 84.05%, test macro 83.92%
Epoch 67: loss 0.00051, train micro 100.0%, train macro 100.0%, valid micro 9
5.67%, valid macro 95.67%, test micro 84.09%, test macro 83.98%
Epoch 68: loss 0.0005, train micro 100.0%, train macro 100.0%, valid micro 9
5.67%, valid macro 95.67%, test micro 84.0%, test macro 83.89%
Epoch 69: loss 0.00049, train micro 100.0%, train macro 100.0%, valid micro 9
5.67%, valid macro 95.67%, test micro 84.09%, test macro 83.99%
Epoch 70: loss 0.00048, train micro 100.0%, train macro 100.0%, valid micro 9
5.67%, valid macro 95.67%, test micro 84.0%, test macro 83.91%
Epoch 71: loss 0.00047, train micro 100.0%, train macro 100.0%, valid micro 9
6.0%, valid macro 96.01%, test micro 84.05%, test macro 83.96%
Epoch 72: loss 0.00047, train micro 100.0%, train macro 100.0%, valid micro 9
6.0%, valid macro 96.01%, test micro 83.91%, test macro 83.82%
Epoch 73: loss 0.00046, train micro 100.0%, train macro 100.0%, valid micro 9
6.0%, valid macro 96.01%, test micro 83.91%, test macro 83.82%
Epoch 74: loss 0.00045, train micro 100.0%, train macro 100.0%, valid micro 9
6.0%, valid macro 96.01%, test micro 83.91%, test macro 83.82%
Epoch 75: loss 0.00044, train micro 100.0%, train macro 100.0%, valid micro 9
6.0%, valid macro 96.01%, test micro 83.95%, test macro 83.88%
Epoch 76: loss 0.00044, train micro 100.0%, train macro 100.0%, valid micro 9
```

```
6.0%, valid macro 96.01%, test micro 84.09%, test macro 84.03%
Epoch 77: loss 0.00043, train micro 100.0%, train macro 100.0%, valid micro 9
6.0%, valid macro 96.01%, test micro 84.05%, test macro 83.99%
Epoch 78: loss 0.00043, train micro 100.0%, train macro 100.0%, valid micro 9
6.33%, valid macro 96.34%, test micro 84.05%, test macro 83.99%
Epoch 79: loss 0.00042, train micro 100.0%, train macro 100.0%, valid micro 9
6.33%, valid macro 96.34%, test micro 84.05%, test macro 83.99%
Epoch 80: loss 0.00041, train micro 100.0%, train macro 100.0%, valid micro 9
6.33%, valid macro 96.34%, test micro 84.0%, test macro 83.95%
Epoch 81: loss 0.00041, train micro 100.0%, train macro 100.0%, valid micro 9
6.67%, valid macro 96.68%, test micro 83.91%, test macro 83.85%
Epoch 82: loss 0.0004, train micro 100.0%, train macro 100.0%, valid micro 9
6.67%, valid macro 96.68%, test micro 84.0%, test macro 83.95%
Epoch 83: loss 0.0004, train micro 100.0%, train macro 100.0%, valid micro 9
6.67%, valid macro 96.68%, test micro 83.86%, test macro 83.81%
Epoch 84: loss 0.00039, train micro 100.0%, train macro 100.0%, valid micro 9
6.67%, valid macro 96.68%, test micro 83.91%, test macro 83.87%
Epoch 85: loss 0.00039, train micro 100.0%, train macro 100.0%, valid micro 9
6.67%, valid macro 96.68%, test micro 83.91%, test macro 83.87%
Epoch 86: loss 0.00039, train micro 100.0%, train macro 100.0%, valid micro 9
6.67%, valid macro 96.68%, test micro 83.91%, test macro 83.87%
Epoch 87: loss 0.00038, train micro 100.0%, train macro 100.0%, valid micro 9
7.0%, valid macro 97.01%, test micro 83.86%, test macro 83.83%
Epoch 88: loss 0.00038, train micro 100.0%, train macro 100.0%, valid micro 9
7.0%, valid macro 97.01%, test micro 83.81%, test macro 83.78%
Epoch 89: loss 0.00037, train micro 100.0%, train macro 100.0%, valid micro 9
7.0%, valid macro 97.01%, test micro 83.86%, test macro 83.83%
Epoch 90: loss 0.00037, train micro 100.0%, train macro 100.0%, valid micro 9
7.0%, valid macro 97.01%, test micro 83.81%, test macro 83.78%
Epoch 91: loss 0.00036, train micro 100.0%, train macro 100.0%, valid micro 9
7.0%, valid macro 97.01%, test micro 83.81%, test macro 83.78%
Epoch 92: loss 0.00036, train micro 100.0%, train macro 100.0%, valid micro 9
7.0%, valid macro 97.01%, test micro 83.81%, test macro 83.78%
Epoch 93: loss 0.00036, train micro 100.0%, train macro 100.0%, valid micro 9
7.0%, valid macro 97.01%, test micro 83.81%, test macro 83.78%
Epoch 94: loss 0.00035, train micro 100.0%, train macro 100.0%, valid micro 9
7.0%, valid macro 97.01%, test micro 83.76%, test macro 83.74%
Epoch 95: loss 0.00035, train micro 100.0%, train macro 100.0%, valid micro 9
7.0%, valid macro 97.01%, test micro 83.58%, test macro 83.57%
Epoch 96: loss 0.00035, train micro 100.0%, train macro 100.0%, valid micro 9
7.0%, valid macro 97.01%, test micro 83.58%, test macro 83.57%
Epoch 97: loss 0.00034, train micro 100.0%, train macro 100.0%, valid micro 9
7.0%, valid macro 97.01%, test micro 83.58%, test macro 83.57%
Epoch 98: loss 0.00034, train micro 100.0%, train macro 100.0%, valid micro 9
7.0%, valid macro 97.01%, test micro 83.58%, test macro 83.57%
Epoch 99: loss 0.00034, train micro 100.0%, train macro 100.0%, valid micro 9
7.0%, valid macro 97.01%, test micro 83.58%, test macro 83.58%
Epoch 100: loss 0.00033, train micro 100.0%, train macro 100.0%, valid micro
97.0%, valid macro 97.01%, test micro 83.58%, test macro 83.57%
Saving Heterogeneous Node Prediction Model Predictions with Agg: Mean

Best model: train micro 100.0%, train macro 100.0%, valid micro 98.0%, valid
macro 98.0%, test micro 86.68%, test macro 86.48%
```

Question 2.1: What is your maximum test set **micro** F1 score for the best_model when using mean aggregation? (10 points)

86.68%

Question 2.2: What is your maximum test set **macro** F1 score for the best_model when using the mean aggregation? (10 points)

86.48%

Training the Attention Aggregation

Start of Debugging

In [40]:
```
hetero_graph.node_feature
hetero_graph.edge_index
```

Out[40]:
```
{('paper',
  'author',
  'paper'): SparseTensor(row=tensor([   0,    0,    0,  ..., 3024, 3024, 302
4]),
              col=tensor([   8,   20,   51,  ..., 2948, 2983, 2991]),
              size=(3025, 3025), nnz=26256, density=0.29%),
 ('paper',
  'subject',
  'paper'): SparseTensor(row=tensor([   0,    0,    0,  ..., 3024, 3024, 302
4]),
              col=tensor([  75,  434,  534,  ..., 3020, 3021, 3022]),
              size=(3025, 3025), nnz=2207736, density=24.13%)}
```

In [41]:
```
for key in hetero_graph.node_feature:
    print(key)
hetero_graph.node_feature['paper'].shape
#hetero_graph.node_feature['author'].shape
#hetero_graph.node_feature['paper'].shape
```

```
paper
```

Out[41]:
```
torch.Size([3025, 1870])
```

In [42]:
```
for key in hetero_graph.edge_index:
    print(key)
    print(hetero_graph.edge_index[key].dense_shape())
```

```
('paper', 'author', 'paper')

---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Input In [42], in <cell line: 1>()
      1 for key in hetero_graph.edge_index:
      2     print(key)
----> 3     print(hetero_graph.edge_index[key].dense_shape())

AttributeError: 'SparseTensor' object has no attribute 'dense_shape'
```

In [43]:
```
print("arguments to HeteroGNN:{}".format(args))
output_size = hetero_graph.num_node_labels('paper')
print("output_size:{}".format(output_size))
```

```
arguments to HeteroGNN:{'device': device(type='cpu'), 'hidden_size': 64, 'epo
chs': 100, 'weight_decay': 1e-05, 'lr': 0.003, 'attn_size': 32}
output_size:3
```

End of Debugging

In [93]:
```
best_model = None
best_val = 0

output_size = hetero_graph.num_node_labels('paper')
model = HeteroGNN(hetero_graph, args, aggr="attn").to(args['device'])
```

```
optimizer = torch.optim.Adam(model.parameters(), lr=args['lr'], weight_decay=

for epoch in range(args['epochs']):

    loss = train(model, optimizer, hetero_graph, train_idx)

    accs, best_model, best_val = test(model, hetero_graph, [train_idx, val_id
    print(
          f"Epoch {epoch + 1}: loss {round(loss, 5)}, "
          f"train micro {round(accs[0][0] * 100, 2)}%, train macro {round(acc
          f"valid micro {round(accs[1][0] * 100, 2)}%, valid macro {round(acc
          f"test micro {round(accs[2][0] * 100, 2)}%, test macro {round(accs[
    )
best_accs, _, _ = test(best_model, hetero_graph, [train_idx, val_idx, test_id
print(
          f"Best model: "
          f"train micro {round(best_accs[0][0] * 100, 2)}%, train macro {rour
          f"valid micro {round(best_accs[1][0] * 100, 2)}%, valid macro {rour
          f"test micro {round(best_accs[2][0] * 100, 2)}%, test macro {round(
)
```

```
Epoch 1: loss 1.10119, train micro 33.33%, train macro 16.67%, valid micro 3
3.33%, valid macro 16.67%, test micro 35.81%, test macro 17.58%
Epoch 2: loss 1.0915, train micro 65.67%, train macro 54.69%, valid micro 65.
33%, valid macro 54.58%, test micro 63.06%, test macro 52.85%
Epoch 3: loss 1.06019, train micro 66.33%, train macro 54.52%, valid micro 6
6.0%, valid macro 54.73%, test micro 65.36%, test macro 54.17%
Epoch 4: loss 0.99695, train micro 66.33%, train macro 54.12%, valid micro 6
6.33%, valid macro 54.43%, test micro 65.6%, test macro 54.08%
Epoch 5: loss 0.88948, train micro 66.17%, train macro 53.74%, valid micro 6
6.33%, valid macro 54.1%, test micro 65.69%, test macro 53.79%
Epoch 6: loss 0.7381, train micro 66.33%, train macro 53.87%, valid micro 66.
33%, valid macro 53.93%, test micro 65.51%, test macro 53.36%
Epoch 7: loss 0.57109, train micro 67.83%, train macro 56.63%, valid micro 6
6.67%, valid macro 54.57%, test micro 65.46%, test macro 53.34%
Epoch 8: loss 0.42575, train micro 69.33%, train macro 59.87%, valid micro 6
7.33%, valid macro 56.18%, test micro 65.36%, test macro 53.49%
Epoch 9: loss 0.31329, train micro 72.33%, train macro 65.4%, valid micro 68.
33%, valid macro 58.27%, test micro 65.41%, test macro 53.89%
Epoch 10: loss 0.22919, train micro 75.5%, train macro 70.75%, valid micro 7
0.0%, valid macro 61.49%, test micro 65.84%, test macro 54.67%
Epoch 11: loss 0.16761, train micro 79.33%, train macro 76.33%, valid micro 7
3.33%, valid macro 67.32%, test micro 66.35%, test macro 55.73%
Epoch 12: loss 0.12285, train micro 84.0%, train macro 82.41%, valid micro 7
7.0%, valid macro 73.0%, test micro 67.01%, test macro 57.24%
Epoch 13: loss 0.09001, train micro 89.33%, train macro 88.76%, valid micro 8
0.0%, valid macro 77.37%, test micro 67.76%, test macro 58.94%
Epoch 14: loss 0.06598, train micro 91.83%, train macro 91.57%, valid micro 8
1.67%, valid macro 79.74%, test micro 68.42%, test macro 60.7%
Epoch 15: loss 0.04865, train micro 93.0%, train macro 92.85%, valid micro 8
4.0%, valid macro 82.73%, test micro 69.22%, test macro 62.49%
Epoch 16: loss 0.03649, train micro 95.83%, train macro 95.78%, valid micro 8
5.33%, valid macro 84.35%, test micro 70.45%, test macro 64.59%
Epoch 17: loss 0.02804, train micro 97.33%, train macro 97.32%, valid micro 8
7.0%, valid macro 86.31%, test micro 71.62%, test macro 66.53%
Epoch 18: loss 0.02212, train micro 98.5%, train macro 98.5%, valid micro 87.
67%, valid macro 87.08%, test micro 73.27%, test macro 69.06%
Epoch 19: loss 0.01755, train micro 99.0%, train macro 99.0%, valid micro 90.
0%, valid macro 89.65%, test micro 74.16%, test macro 70.42%
Epoch 20: loss 0.01407, train micro 99.5%, train macro 99.5%, valid micro 92.
33%, valid macro 92.17%, test micro 75.48%, test macro 72.28%
Epoch 21: loss 0.01136, train micro 99.67%, train macro 99.67%, valid micro 9
3.67%, valid macro 93.59%, test micro 77.08%, test macro 74.47%
Epoch 22: loss 0.00923, train micro 100.0%, train macro 100.0%, valid micro 9
4.0%, valid macro 93.93%, test micro 78.26%, test macro 76.02%
Epoch 23: loss 0.00753, train micro 100.0%, train macro 100.0%, valid micro 9
4.67%, valid macro 94.61%, test micro 79.11%, test macro 77.14%
```

```
Epoch 24: loss 0.00589, train micro 100.0%, train macro 100.0%, valid micro 9
5.0%, valid macro 94.96%, test micro 80.09%, test macro 78.4%
Epoch 25: loss 0.00465, train micro 100.0%, train macro 100.0%, valid micro 9
5.0%, valid macro 94.96%, test micro 80.94%, test macro 79.38%
Epoch 26: loss 0.00371, train micro 100.0%, train macro 100.0%, valid micro 9
6.33%, valid macro 96.32%, test micro 81.32%, test macro 79.85%
Epoch 27: loss 0.00302, train micro 100.0%, train macro 100.0%, valid micro 9
6.33%, valid macro 96.32%, test micro 81.6%, test macro 80.24%
Epoch 28: loss 0.0025, train micro 100.0%, train macro 100.0%, valid micro 9
6.67%, valid macro 96.66%, test micro 82.07%, test macro 80.78%
Epoch 29: loss 0.00211, train micro 100.0%, train macro 100.0%, valid micro 9
6.67%, valid macro 96.66%, test micro 82.54%, test macro 81.38%
Epoch 30: loss 0.00182, train micro 100.0%, train macro 100.0%, valid micro 9
6.67%, valid macro 96.66%, test micro 82.64%, test macro 81.54%
Epoch 31: loss 0.0016, train micro 100.0%, train macro 100.0%, valid micro 9
6.67%, valid macro 96.66%, test micro 82.87%, test macro 81.86%
Epoch 32: loss 0.00142, train micro 100.0%, train macro 100.0%, valid micro 9
7.0%, valid macro 97.0%, test micro 82.82%, test macro 81.83%
Epoch 33: loss 0.00127, train micro 100.0%, train macro 100.0%, valid micro 9
7.0%, valid macro 97.0%, test micro 82.87%, test macro 81.92%
Epoch 34: loss 0.00115, train micro 100.0%, train macro 100.0%, valid micro 9
7.0%, valid macro 97.0%, test micro 83.01%, test macro 82.11%
Epoch 35: loss 0.00105, train micro 100.0%, train macro 100.0%, valid micro 9
7.0%, valid macro 97.0%, test micro 83.01%, test macro 82.17%
Epoch 36: loss 0.00096, train micro 100.0%, train macro 100.0%, valid micro 9
7.0%, valid macro 97.0%, test micro 82.87%, test macro 82.03%
Epoch 37: loss 0.00089, train micro 100.0%, train macro 100.0%, valid micro 9
7.33%, valid macro 97.33%, test micro 82.87%, test macro 82.07%
Epoch 38: loss 0.00082, train micro 100.0%, train macro 100.0%, valid micro 9
8.0%, valid macro 98.0%, test micro 82.78%, test macro 82.04%
Epoch 39: loss 0.00076, train micro 100.0%, train macro 100.0%, valid micro 9
7.67%, valid macro 97.66%, test micro 82.82%, test macro 82.1%
Epoch 40: loss 0.00071, train micro 100.0%, train macro 100.0%, valid micro 9
7.33%, valid macro 97.33%, test micro 82.82%, test macro 82.14%
Epoch 41: loss 0.00067, train micro 100.0%, train macro 100.0%, valid micro 9
7.33%, valid macro 97.33%, test micro 83.01%, test macro 82.41%
Epoch 42: loss 0.00063, train micro 100.0%, train macro 100.0%, valid micro 9
7.33%, valid macro 97.33%, test micro 83.15%, test macro 82.56%
Epoch 43: loss 0.00059, train micro 100.0%, train macro 100.0%, valid micro 9
7.67%, valid macro 97.66%, test micro 83.2%, test macro 82.65%
Epoch 44: loss 0.00056, train micro 100.0%, train macro 100.0%, valid micro 9
8.33%, valid macro 98.33%, test micro 83.29%, test macro 82.81%
Epoch 45: loss 0.00053, train micro 100.0%, train macro 100.0%, valid micro 9
8.33%, valid macro 98.33%, test micro 83.25%, test macro 82.8%
Epoch 46: loss 0.00051, train micro 100.0%, train macro 100.0%, valid micro 9
8.33%, valid macro 98.33%, test micro 83.44%, test macro 83.02%
Epoch 47: loss 0.00048, train micro 100.0%, train macro 100.0%, valid micro 9
8.33%, valid macro 98.33%, test micro 83.44%, test macro 83.04%
Epoch 48: loss 0.00046, train micro 100.0%, train macro 100.0%, valid micro 9
8.33%, valid macro 98.33%, test micro 83.25%, test macro 82.87%
Epoch 49: loss 0.00044, train micro 100.0%, train macro 100.0%, valid micro 9
8.33%, valid macro 98.33%, test micro 83.25%, test macro 82.89%
Epoch 50: loss 0.00043, train micro 100.0%, train macro 100.0%, valid micro 9
8.33%, valid macro 98.33%, test micro 83.25%, test macro 82.9%
Epoch 51: loss 0.00041, train micro 100.0%, train macro 100.0%, valid micro 9
8.33%, valid macro 98.33%, test micro 83.39%, test macro 83.06%
Epoch 52: loss 0.0004, train micro 100.0%, train macro 100.0%, valid micro 9
8.33%, valid macro 98.33%, test micro 83.44%, test macro 83.13%
Epoch 53: loss 0.00038, train micro 100.0%, train macro 100.0%, valid micro 9
8.0%, valid macro 98.0%, test micro 83.34%, test macro 83.04%
Epoch 54: loss 0.00037, train micro 100.0%, train macro 100.0%, valid micro 9
8.0%, valid macro 98.0%, test micro 83.15%, test macro 82.85%
Epoch 55: loss 0.00036, train micro 100.0%, train macro 100.0%, valid micro 9
8.0%, valid macro 98.0%, test micro 83.11%, test macro 82.8%
Epoch 56: loss 0.00035, train micro 100.0%, train macro 100.0%, valid micro 9
8.0%, valid macro 98.0%, test micro 82.87%, test macro 82.59%
Epoch 57: loss 0.00034, train micro 100.0%, train macro 100.0%, valid micro 9
8.0%, valid macro 98.0%, test micro 82.87%, test macro 82.6%
Epoch 58: loss 0.00033, train micro 100.0%, train macro 100.0%, valid micro 9
```

```
8.0%, valid macro 98.0%, test micro 82.73%, test macro 82.48%
Epoch 59: loss 0.00032, train micro 100.0%, train macro 100.0%, valid micro 9
8.0%, valid macro 98.0%, test micro 82.68%, test macro 82.44%
Epoch 60: loss 0.00031, train micro 100.0%, train macro 100.0%, valid micro 9
8.0%, valid macro 98.0%, test micro 82.68%, test macro 82.44%
Epoch 61: loss 0.00031, train micro 100.0%, train macro 100.0%, valid micro 9
8.0%, valid macro 98.0%, test micro 82.73%, test macro 82.49%
Epoch 62: loss 0.0003, train micro 100.0%, train macro 100.0%, valid micro 9
8.0%, valid macro 98.0%, test micro 82.64%, test macro 82.39%
Epoch 63: loss 0.00029, train micro 100.0%, train macro 100.0%, valid micro 9
8.0%, valid macro 98.0%, test micro 82.59%, test macro 82.35%
Epoch 64: loss 0.00029, train micro 100.0%, train macro 100.0%, valid micro 9
8.0%, valid macro 98.0%, test micro 82.45%, test macro 82.21%
Epoch 65: loss 0.00028, train micro 100.0%, train macro 100.0%, valid micro 9
8.0%, valid macro 98.0%, test micro 82.35%, test macro 82.12%
Epoch 66: loss 0.00028, train micro 100.0%, train macro 100.0%, valid micro 9
8.0%, valid macro 98.0%, test micro 82.21%, test macro 81.98%
Epoch 67: loss 0.00027, train micro 100.0%, train macro 100.0%, valid micro 9
8.0%, valid macro 98.0%, test micro 82.26%, test macro 82.03%
Epoch 68: loss 0.00027, train micro 100.0%, train macro 100.0%, valid micro 9
8.0%, valid macro 98.0%, test micro 82.26%, test macro 82.03%
Epoch 69: loss 0.00026, train micro 100.0%, train macro 100.0%, valid micro 9
8.0%, valid macro 98.0%, test micro 82.16%, test macro 81.95%
Epoch 70: loss 0.00026, train micro 100.0%, train macro 100.0%, valid micro 9
8.0%, valid macro 98.0%, test micro 82.16%, test macro 81.95%
Epoch 71: loss 0.00025, train micro 100.0%, train macro 100.0%, valid micro 9
8.0%, valid macro 98.0%, test micro 82.26%, test macro 82.06%
Epoch 72: loss 0.00025, train micro 100.0%, train macro 100.0%, valid micro 9
8.0%, valid macro 98.0%, test micro 82.16%, test macro 81.96%
Epoch 73: loss 0.00024, train micro 100.0%, train macro 100.0%, valid micro 9
8.0%, valid macro 98.0%, test micro 82.12%, test macro 81.92%
Epoch 74: loss 0.00024, train micro 100.0%, train macro 100.0%, valid micro 9
8.0%, valid macro 98.0%, test micro 82.12%, test macro 81.92%
Epoch 75: loss 0.00024, train micro 100.0%, train macro 100.0%, valid micro 9
8.0%, valid macro 98.0%, test micro 82.12%, test macro 81.92%
Epoch 76: loss 0.00023, train micro 100.0%, train macro 100.0%, valid micro 9
8.0%, valid macro 98.0%, test micro 82.07%, test macro 81.87%
Epoch 77: loss 0.00023, train micro 100.0%, train macro 100.0%, valid micro 9
8.0%, valid macro 98.0%, test micro 82.07%, test macro 81.87%
Epoch 78: loss 0.00023, train micro 100.0%, train macro 100.0%, valid micro 9
8.0%, valid macro 98.0%, test micro 82.12%, test macro 81.93%
Epoch 79: loss 0.00022, train micro 100.0%, train macro 100.0%, valid micro 9
8.0%, valid macro 98.0%, test micro 82.12%, test macro 81.93%
Epoch 80: loss 0.00022, train micro 100.0%, train macro 100.0%, valid micro 9
8.0%, valid macro 98.0%, test micro 82.12%, test macro 81.93%
Epoch 81: loss 0.00022, train micro 100.0%, train macro 100.0%, valid micro 9
8.0%, valid macro 98.0%, test micro 82.02%, test macro 81.84%
Epoch 82: loss 0.00022, train micro 100.0%, train macro 100.0%, valid micro 9
8.0%, valid macro 98.0%, test micro 82.02%, test macro 81.84%
Epoch 83: loss 0.00021, train micro 100.0%, train macro 100.0%, valid micro 9
8.0%, valid macro 98.0%, test micro 82.02%, test macro 81.84%
Epoch 84: loss 0.00021, train micro 100.0%, train macro 100.0%, valid micro 9
8.0%, valid macro 98.0%, test micro 82.02%, test macro 81.84%
Epoch 85: loss 0.00021, train micro 100.0%, train macro 100.0%, valid micro 9
8.0%, valid macro 98.0%, test micro 82.02%, test macro 81.84%
Epoch 86: loss 0.00021, train micro 100.0%, train macro 100.0%, valid micro 9
8.0%, valid macro 98.0%, test micro 82.02%, test macro 81.84%
Epoch 87: loss 0.0002, train micro 100.0%, train macro 100.0%, valid micro 9
8.0%, valid macro 98.0%, test micro 82.07%, test macro 81.89%
Epoch 88: loss 0.0002, train micro 100.0%, train macro 100.0%, valid micro 9
8.0%, valid macro 98.0%, test micro 82.02%, test macro 81.85%
Epoch 89: loss 0.0002, train micro 100.0%, train macro 100.0%, valid micro 9
8.0%, valid macro 98.0%, test micro 82.02%, test macro 81.85%
Epoch 90: loss 0.0002, train micro 100.0%, train macro 100.0%, valid micro 9
8.0%, valid macro 98.0%, test micro 81.93%, test macro 81.76%
Epoch 91: loss 0.00019, train micro 100.0%, train macro 100.0%, valid micro 9
8.0%, valid macro 98.0%, test micro 81.93%, test macro 81.76%
Epoch 92: loss 0.00019, train micro 100.0%, train macro 100.0%, valid micro 9
8.0%, valid macro 98.0%, test micro 81.93%, test macro 81.76%
```

```
Epoch 93: loss 0.00019, train micro 100.0%, train macro 100.0%, valid micro 98.0%, valid macro 98.0%, test micro 81.93%, test macro 81.76%
Epoch 94: loss 0.00019, train micro 100.0%, train macro 100.0%, valid micro 98.0%, valid macro 98.0%, test micro 81.93%, test macro 81.76%
Epoch 95: loss 0.00019, train micro 100.0%, train macro 100.0%, valid micro 98.0%, valid macro 98.0%, test micro 81.93%, test macro 81.76%
Epoch 96: loss 0.00018, train micro 100.0%, train macro 100.0%, valid micro 98.0%, valid macro 98.0%, test micro 81.84%, test macro 81.67%
Epoch 97: loss 0.00018, train micro 100.0%, train macro 100.0%, valid micro 98.0%, valid macro 98.0%, test micro 81.79%, test macro 81.63%
Epoch 98: loss 0.00018, train micro 100.0%, train macro 100.0%, valid micro 98.0%, valid macro 98.0%, test micro 81.79%, test macro 81.63%
Epoch 99: loss 0.00018, train micro 100.0%, train macro 100.0%, valid micro 98.0%, valid macro 98.0%, test micro 81.79%, test macro 81.63%
Epoch 100: loss 0.00018, train micro 100.0%, train macro 100.0%, valid micro 98.0%, valid macro 98.0%, test micro 81.84%, test macro 81.68%

Best model: train micro 100.0%, train macro 100.0%, valid micro 98.33%, valid macro 98.33%, test micro 83.29%, test macro 82.81%
```

In [95]:

```
# if 'IS_GRADESCOPE_ENV' not in os.environ:
#     best_model = None
#     best_val = 0

#     output_size = hetero_graph.num_node_labels('paper')
#     model = HeteroGNN(hetero_graph, args, aggr="attn").to(args['device'])
#     optimizer = torch.optim.Adam(model.parameters(), lr=args['lr'], weight_

#   #for epoch in range(args['epochs']):

#     loss = train(model, optimizer, hetero_graph, train_idx)
#     accs, best_model, best_val = test(model, hetero_graph, [train_idx, val_
#     print(
#           f"Epoch {epoch + 1}: loss {round(loss, 5)}, "
#           f"train micro {round(accs[0][0] * 100, 2)}%, train macro {round(a
#           f"valid micro {round(accs[1][0] * 100, 2)}%, valid macro {round(a
#           f"test micro {round(accs[2][0] * 100, 2)}%, test macro {round(acc
#     )
#     best_accs, _, _ = test(best_model, hetero_graph, [train_idx, val_idx, t
#     print(
#           f"Best model: "
#           f"train micro {round(best_accs[0][0] * 100, 2)}%, train macro {ro
#           f"valid micro {round(best_accs[1][0] * 100, 2)}%, valid macro {ro
#           f"test micro {round(best_accs[2][0] * 100, 2)}%, test macro {rour
#     )
```

Question 2.3: What is your maximum test set **micro** F1 score for the best_model when using the attention aggregation? (4 points)

83.29%

Question 2.4: What is your maximum test set **macro** F1 score for the best_model when using the attention aggregation? (4 points)

82.81%

Attention for each Message Type

Through message type level attention we can learn which message type is more important to which layer.

Here we will print out and show that each layer pay how much attention on each message type.

In [94]:
```
if 'IS_GRADESCOPE_ENV' not in os.environ:
  if model.convs1.alpha is not None and model.convs2.alpha is not None:
      for idx, message_type in model.convs1.mapping.items():
          print(f"Layer 1 has attention {model.convs1.alpha[idx]} on message
      for idx, message_type in model.convs2.mapping.items():
          print(f"Layer 2 has attention {model.convs2.alpha[idx]} on message
```

```
Layer 1 has attention 0.09191861003637314 on message type ('paper', 'author',
'paper')
Layer 1 has attention 0.9080814123153687 on message type ('paper', 'subject',
'paper')
Layer 2 has attention 0.9286372661590576 on message type ('paper', 'author',
'paper')
Layer 2 has attention 0.07136277854442596 on message type ('paper', 'subjec
t', 'paper')
```

Submission

You will need to submit three files on Gradescope to complete this notebook.

1. Your completed *CS224W_Colab5.ipynb*. From the "File" menu select "Download .ipynb" to save a local copy of your completed Colab.
2. *ACM-Node-MeanAgg.csv*
3. *ACM-Node-AttentionAgg.csv*

Download the csv files by selecting the *Folder* icon on the left panel.

To submit your work, zip the files downloaded in steps 1-3 above and submit to gradescope. **NOTE:** DO NOT rename any of the downloaded files.