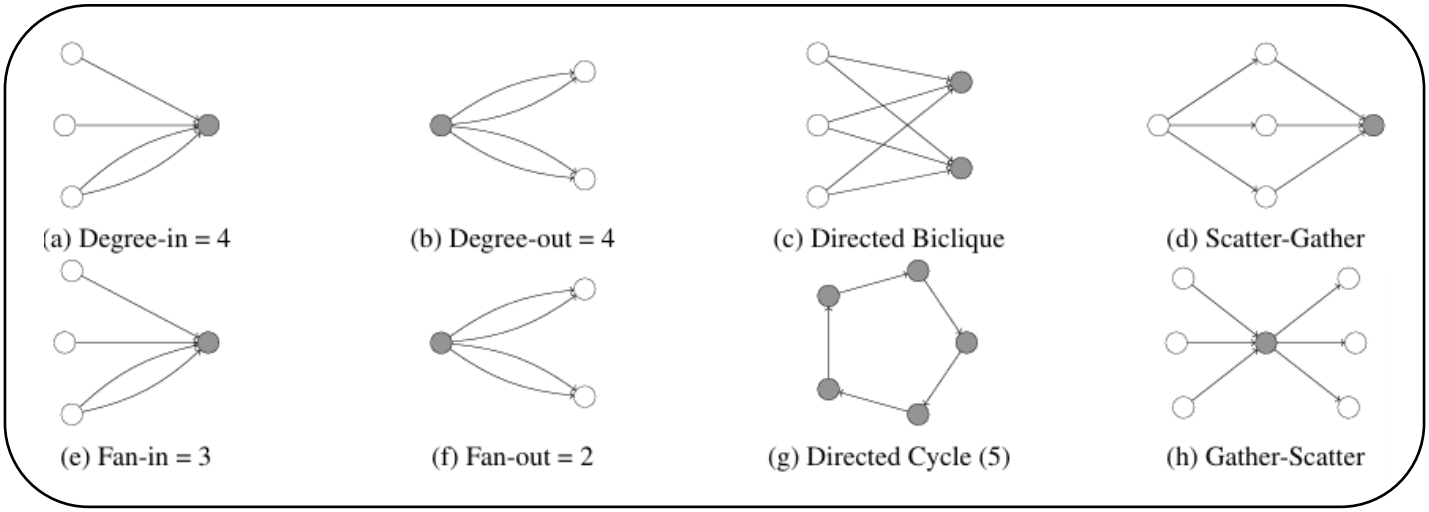
Multigraph GNN for Anti-money Laundering

Abstract: The paper analyses a set of simple adaptations that can help transform Standard Message - Passing Graph Neural Networks (GNN) into provably powerful directed multigraph neural networks. ***The adaptations include multigraph port numbering, ego IDs, and reverse message passing.***

A multigraph is a graph in which there can be more than one edge between two nodes, which is a very common instance in a transactional data, as there can be more than one transaction between two nodes (accounts).

The paper demonstrates that these adaptations together theoretically enable the GNN to detect any directed sub-graph pattern. This is, to our knowledge, the first GNN architecture explicitly designed for directed multigraphs.

We observe dramatic improvements in detecting Money Laundering transactions, improving the minority-class F1 score of a standard message-passing GNN by up to 30%, and closely matching or outperforming tree-based and GNN baselines.



ML Patterns:

The synthetic pattern detection dataset includes the following (sub-)tasks:

• Degree-in (-out): Degree-in and -out of a node is the number of incoming and outgoing transactions, respectively. Note that multiple transactions can come from the same node, and these are counted separately.

• Fan-in (-out): Fan-in and -out of a node are the number of unique incoming and outgoing neighbors, respectively. Neighbours with multiple edges are counted only once, but a node can contribute to both the fan-in and fan-out counts, as it can be both an incoming and outgoing neighbor.

• Cycle (k): A node receives a positive label if it is part of a directed cycle of length k. We only consider directed cycle patterns and omit the word “directed.”

• Scatter-gather: A node receives a positive label if it is the sink node of a scatter-gather pattern with at least 2 intermediate nodes. A scatter-gather pattern consists of a source node, intermediate nodes, and a sink node. Given a sink node v, a node u is considered a source node if there is a directed path of length 2 from u to v. A node wi is considered an intermediate node if there is a directed path of length 2 from u to v through wi , i.e., if both (u, wi) and (wi , v) are in E(G). Note that a scatter-gather pattern with a single intermediate node is simply a directed path of length 2. This does not capture the pattern we are interested in, so we require at least 2 intermediate nodes.

• Gather-Scatter: We do not explicitly include this pattern as a subtask since it is a combination of fan-in and fan-out.

• Biclique: A biclique (or complete bipartite graph) is a bipartite graph where every node of the first set is connected to every node of the second set. We consider a special case of the directed biclique, where every node in the first set has at least one outgoing edge to every node in the second set. We call the nodes in the first set source nodes and the nodes in the second set sink nodes. The task is to identify all nodes that are the sink nodes of some directed K2,2 biclique with (at least) 2 source nodes and (at least) 2 sink nodes.

Synthetic Pattern Data and its generation:

The synthetically generated data can be found at: [*Synthetic Data for Anti Money Laundering*]

The AML subgraph patterns seen in Fig. 1 are used to create a controllable testbed of synthetic pattern detection tasks. The key design principle is to ensure that the desired subgraph patterns appear randomly, rather than being inserted post hoc into a graph. To ensure that the desired subgraph patterns appear randomly, we introduce the **random circulant graph** generator.

The generator is set up with three parameters: the number of nodes n, the average degree d, and a radius r that controls the local density of the graph. These parameters determine the rate at which different subgraph patterns are generated. For instance, by selecting a high average degree and a low radius, we can ensure that denser patterns appear more frequently.

An RC graph $G_{n,d,r}$ is created with n nodes labeled {1,2,...,n} and ⌊nd/2⌋ edges. Each edge connects two nodes, u and v. Here's how they are chosen:

1. The first node, u, is picked randomly from {1,2,...,n}.
2. The second node, v, is chosen from a normal distribution centered around u with a standard deviation r, and then rounded to the nearest integer, i.e., using v = ⌊X+1/2⌋, where X~N(u,r).

This model is inspired from the Watts–Strogatz small-world model (Watts and Strogatz 1998), where nodes are also arranged in a ring, each node is connected to its k nearest neighbors, and edges are randomly rewired with a chosen probability.

Algorithm 2: Random Circulant Graph Generator

Parameters: Number of nodes n, average degree d, locality radius r.
Output: A graph $G = (V, E)$.

$N_{edges} \leftarrow \lfloor \frac{nd}{2} \rfloor$
$\vec{u} \leftarrow (u_i \mid i = 1, \ldots, N_{edges}) \overset{\text{iid}}{\sim} \text{unif}(\{1, \ldots, n\})$ ▷ Sample tail nodes
$\vec{v} \leftarrow (v_i \mid i = 1, \ldots, N_{edges}) \overset{\text{iid}}{\sim} \mathcal{N}(u_i, r)$ ▷ Sample head nodes
$\vec{v} \leftarrow (\lfloor v_i + \frac{1}{2} \rfloor \mid i = 1, \ldots, N_{edges})$ ▷ Round head nodes to nearest integer
$V(G) = \{1, \ldots, n\}$
$E(G) = \{(u_i, v_i) \mid i = 1, \ldots, N_{edges}\}$

Dataset	# nodes	# edges	Illicit Ratio
AML Small HI	0.5M	5M	0.07%
AML Small LI	0.7M	7M	0.05%
AML Medium HI	2.1M	32M	0.11%
AML Medium LI	2.0M	31M	0.05%
ETH	2.9M	13M	0.04%

Nodes and Edges:

We consider directed multigraphs, G, where the nodes v ∈ V (G) represent accounts, and the directed edges e = (u, v) ∈ E(G) represent transactions from u to v. Each node u (optionally) has a set of account features h (0)(u); this could include **the account number, bank ID, and account balance.** Each transaction e = (u, v) has a set of associated transaction features $h^{(0)}_{(u,v)}$; this includes the **amount, payment format, and timestamp of the transaction.** The incoming and outgoing neighbors of u are denoted by $N_{in}(u)$ and $N_{out}(u)$ respectively.

Message Passing Neural Network (MPNN):

Message-passing GNNs, include GCN, GIN, GAT, GraphSAGE, and many more architectures.

They work in three steps (making a layer of GNN):

(1) Each node sends a message with its current state h(v) to its neighbours,

(2) Each node aggregates all the messages it receives from its neighbours in the embedding a(v), and

(3) Each node updates its state based on h(v) and a(v) to produce a new state.

$$a^{(t)}(v) = \textsc{Aggregate}\left(\{h^{(t-1)}(u) \mid u \in N(v)\}\right),$$
$$h^{(t)}(v) = \textsc{Update}\left(h^{(t-1)}(v), a^{(t)}(v)\right),$$

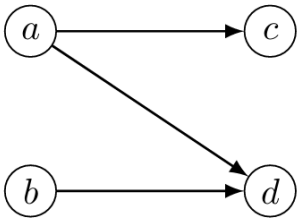
In a standard MPNN, the messages are passed along the directed edges in the direction indicated. Therefore, the aggregation step only considers messages from incoming neighbors:

$$a^{(t)}(v) = \textsc{Agg}\left(\{h^{(t-1)}(u) \mid u \in N_{in}(v)\}\right),$$

where we aggregate over the incoming neighbors, $N_{in}(v)$.

Methods / Adaptations:

Reverse Message Passing: In Standard MPNN with directed edges, a node does not receive any messages from outgoing neighbors (unless they happen to also be incoming neighbors), and so is unable to count its outgoing edges. Example: Its unable to distinguish between a and b. Furthermore, common solution will be to use bidirectional MP, but Naive bidirectional MP, where edges are treated as undirected and messages travel in both directions, cannot distinguish incoming and outgoing edges. So, this would fail to distinguish nodes a and d in the same figure.



To overcome this issue, we need to indicate the direction of the edges in some way. Thus, we propose using a separate message-passing layer for the incoming and outgoing edges, respectively, **i.e., adding reverse message-passing. Thus, we reverse the direction of edges and again implement MP.**

Formally, the aggregation and update mechanisms become:

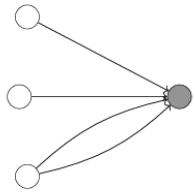
$$a_{in}^{(t)}(v) = \text{AGG}_{in}\left(\{h^{(t-1)}(u) \mid u \in N_{in}(v)\}\right),$$
$$a_{out}^{(t)}(v) = \text{AGG}_{out}\left(\{h^{(t-1)}(u) \mid u \in N_{out}(v)\}\right),$$
$$h^{(t)}(v) = \text{UPDATE}\left(h^{(t-1)}(v), a_{in}^{(t)}(v), a_{out}^{(t)}(v)\right),$$

An MPNN with sum aggregation and reverse MP can solve degree out.

Directed Multigraph Port Numbering:

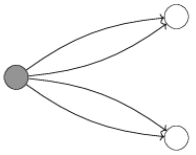
To detect fan-in (or fan-out) patterns, a model must distinguish between edges from the same neighbour and edges from different neighbours. **Using unique account numbers (or in general node IDs) would naturally allow for this. However, using account numbers does not generalize well.** During training, a model can memorize fraudulent account numbers without learning to identify fraudulent patterns.

Therefore, **Port numbering** assigns local IDs to each neighbour at a node. This allows a node to identify messages coming from the same neighbour in consecutive MP rounds. To adapt port numbering to directed multigraphs, we assign each directed edge an incoming and an outgoing port number, and edges coming from (or going to) the same node, receive the same incoming (or outgoing) port number. Moreover, we attach the port numbers in both directions, i.e., a node sees both the port number it has assigned to a neighbor and the port number that the neighbor has assigned to it, as it is crucial to distinguish the centre nodes.



(e) Fan-in = 3

Usually, assigning port numbers to the edges around a node is done randomly. A node with d incoming edges can assign these numbers in d! (factorial) different ways. To make this process consistent in our datasets, we use transaction timestamps to order the incoming (or outgoing) edges. For parallel edges (multiple edges between the same nodes), we use the earliest timestamp to decide the order. This makes sense in financial crime detection since two identical patterns with different timestamps can have different meanings.



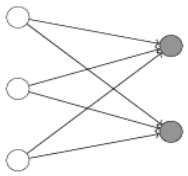
(f) Fan-out = 2

Sorting the edges by timestamps to assign port numbers can be time-consuming, with a runtime complexity of O(mlogm), where m is the number of edges in the graph. However, this step can be done beforehand, so the training and inference times of the model are not affected. Each edge will then have an incoming and outgoing port number as additional features.

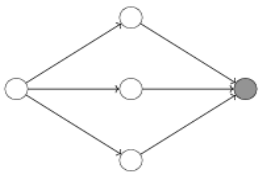
An MPNN with max aggregation and multigraph port numbering can solve fan-in & an MPNN with max aggregation, multigraph port numbering, and reverse MP can solve fan-in and fan-out.

Ego IDs:

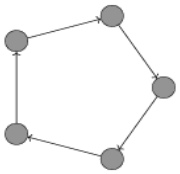
Ego IDs specifically help to detect directed cycles, scatter-gather pat terns, and directed bicliques. The idea is that by marking a "center" node, referred as ego, with a unique (binary) feature, this node can recognize when a sequence of messages cycles back to it, helping detect cycles that it is part of.



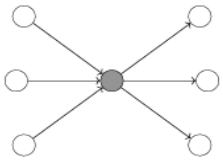
(c) Directed Biclique



(d) Scatter-Gather



(g) Directed Cycle (5)



(h) Gather-Scatter

Ego Id helps a node distinguish between their own features and those of their neighbors within an ego network during updation step. This distinction is crucial for effectively capturing and utilizing local structural information, which is important for tasks like counting cycles and detecting subgraph structures.

In combination with reverse MP and port numbering, ego IDs can detect cycles, scatter-gather patterns, and bipartite subgraphs, completing the list of suspicious patterns.

In fact, it can be shown that a suitably powerful standard MPNN with these adaptations can distinguish any two non-isomorphic (sub)graphs and given a consistent use of port-numbering. GNNs fulfilling these two properties are often referred to as universal.

GNNs with Ego IDs, port numbering, and reverse MP can theoretically detect any directed subgraph pattern.

Time and Complexity:

The final time complexity of a model will depend on the choice of base GNN.

Reverse message passing multiples the time complexity by a constant factor. Theoretically, this constant factor would be 2, but in practice, we also increase the memory requirements of the model by a factor of 2, so the batch size has to be reduced, leading to a larger constant factor.

Port numbering does not increase the time complexity of training and inference since it simply increases the size of the input edge features. However, computing timestamp-based port numbers adds a pre-computation step of complexity O(mlog(m)), which can be done in one go before training.

Ego IDs similarly do not increase the runtime complexity.

Model	TTT (s)	TPS (inference)
GIN	1876	53130
Individual Adaptations		
GIN+Ports	2007	58663
GIN+EgoIDs	1873	58533
GIN+ReverseMP	12510	21067
Cumulative Adaptations		
GIN+ReverseMP+Ports	13051	18763
+EgoIDs	12958	18700

Table 5: Total training times (TTT) and Transaction per Second (TPS) for all GIN-based models on the AML Small HI dataset. The TPS values were measured in evaluation mode.

All in all, the adaptations add a constant factor to the runtime complexity in addition to a pre-computation runtime of complexity O(mlog(m)).

Results:

GIN with edge features is used as the main GNN base model with our adaptations added on top. GAT and PNA are also used as base models, and we refer to their adapted versions as Multi-GAT and Multi-PNA. Since AML is an edge classification problem, we also include a baseline using edge updates, denoted GIN+EU. This approach is similar to replacing edges with nodes and running a GNN on said line graph.

Given the size of the AML and ETH datasets, we use neighborhood sampling for all GNN-based models.

Model	deg-in	deg-out	fan-in	fan-out	C2	C3	C4	C5	C6	S-G	B-C
GIN (Xu et al. 2018; Hu et al. 2019)	99.77	43.58	95.57	35.91	34.67	58.00	50.80	43.12	48.59	69.31	63.12
GAT (Veličković et al. 2017)	10.33	10.53	9.69	0.00	0.00	0.00	25.86	0.00	0.00	0.00	0.00
PNA (Velickovic et al. 2019)	99.63	43.02	95.00	38.93	25.77	54.75	51.92	48.79	48.40	65.88	65.51
GIN+EU (Battaglia et al. 2018)	99.30	42.74	95.70	39.13	32.58	55.91	54.65	47.62	49.68	68.54	64.64
GIN+EgoIDs (You et al. 2021)	99.78	51.48	95.06	49.24	98.13	97.97	53.12	44.37	45.42	66.44	63.90
GIN+Ports (Sato, Yamada, and Kashima 2019)	99.47	45.00	99.59	41.51	27.79	56.11	42.68	41.11	44.99	67.99	65.76
GIN+ReverseMP (Jaume et al. 2019)	98.87	99.08	94.99	95.25	35.96	63.85	69.09	67.44	71.23	65.83	66.18
+Ports	98.41	98.35	98.51	99.16	39.15	63.58	69.00	70.35	75.04	67.84	65.78
+EgoIDs (Multi-GIN)	99.48	99.09	99.62	99.32	98.97	98.73	97.46	91.60	84.23	97.42	94.33
Multi-GAT	98.68	98.36	99.28	99.33	98.61	98.93	98.90	95.82	91.81	96.66	86.92
Multi-PNA	99.64	99.25	99.53	99.41	99.71	99.54	99.49	97.46	88.75	99.07	96.77
Multi-GIN+EU	99.55	99.53	99.76	99.77	99.37	99.71	98.73	95.73	88.38	98.81	97.82

Table 1: Minority class F1 scores (%) for the synthetic subgraph detection tasks. First from the top are the standard MPNN baselines; then the results with each adaptation added separately on top of GIN; followed by GIN with the adaptations added cumulatively; and finally, results for the other GNN baselines with the three adaptations (Multi-GNNs). The Ck abbreviations stand for directed k-cycle detection, S-G stands for scatter-gather and B-C stands for biclique detection. We report minority class F1 scores averaged over five runs. We omit standard deviations in favor of readability.

Model	AML Small HI	AML Small LI	AML Medium HI	AML Medium LI	ETH
LightGBM+GFs (Altman et al. 2023)	62.86 ± 0.25	20.83 ± 1.50	59.48 ± 0.15	20.85 ± 0.38	53.20 ± 0.60
XGBoost+GFs (Altman et al. 2023)	63.23 ± 0.17	27.30 ± 0.33	65.70 ± 0.26	28.16 ± 0.14	49.40 ± 0.54
GIN (Xu et al. 2018; Hu et al. 2019)	28.70 ± 1.13	7.90 ± 2.78	42.20 ± 0.44	3.86 ± 3.62	26.92 ± 7.52
PNA (Velickovic et al. 2019)	56.77 ± 2.41	14.85 ± 1.46	59.71 ± 1.91	27.73 ± 1.65	51.49 ± 4.26
GIN+EU (Battaglia et al. 2018)	47.73 ± 7.86	20.62 ± 2.41	49.26 ± 4.02	6.19 ± 8.32	33.92 ± 7.34
R-GCN (Schlichtkrull et al. 2018)	41.78 ± 0.48	7.43 ± 0.38	OOM	OOM	OOM
GIN+EgoIDs (You et al. 2021)	39.65 ± 4.73	14.98 ± 2.66	45.26 ± 2.16	11.17 ± 6.41	26.01 ± 2.27
GIN+Ports (Sato, Yamada, and Kashima 2019)	54.85 ± 0.89	21.41 ± 2.40	54.22 ± 1.94	10.51 ± 12.82	32.96 ± 0.25
GIN+ReverseMP (Jaume et al. 2019)	46.79 ± 4.97	15.98 ± 4.39	51.93 ± 2.90	14.00 ± 9.34	36.86 ± 8.12
+Ports	56.85 ± 2.64	23.80 ± 4.07	57.15 ± 0.76	11.39 ± 8.36	42.51 ± 7.16
+EgoIDs (Multi-GIN)	57.15 ± 4.99	22.12 ± 2.88	56.23 ± 1.51	14.55 ± 2.91	42.86 ± 2.53
Multi-GIN+EU	64.79 ± 1.22	26.88 ± 6.63	58.92 ± 1.83	16.30 ± 4.73	48.37 ± 6.62
Multi-PNA	64.59 ± 3.60	30.65 ± 2.00	65.67 ± 2.66	33.23 ± 1.31	65.28 ± 2.89
Multi-PNA+EU	68.16 ± 2.65	33.07 ± 2.63	66.48 ± 1.63	36.07 ± 1.17	66.58 ± 1.60

Table 2: Minority class F1 scores (%) for the AML and ETH tasks. HI indicates a higher illicit ratio and LI indicates a lower illicit ratio. The models are organized as in Table 1. "OOM" indicates that the model ran out of GPU memory.

Code Work Flow:

The code is available at: *IBM/Multi-GNN: Multi-GNN architectures for Anti-Money Laundering*

The code initiates the main function and processes the command line arguments according to the specified format.

The ***get_data()*** function retrieves the data in the required format for subsequent steps. The file ***"formatted_transactions.csv"*** is generated by the ***format_kaggle_files.py*** script, which converts raw data into a format containing the exact relevant information.

The columns of this file are:

" EdgeID, from_id, to_id, Timestamp, Amount Sent, Sent Currency, Amount Received, Received Currency, Payment Format, Is Laundering "

The Timestamp is normalized starting from zero. Random IDs are assigned to every edge and node, which are then used to create adjacency lists (features) for each node and edge. The data is converted into tensors to make it trainable by PyTorch models. Dictionaries are created for each node and edge to store all features. The data is then split into training, testing, and validation sets.

The ***GraphData()*** and ***HeteroGraphData()*** classes represent the graph data structure. The training, testing, and validation splits are converted into separate data objects. All the objects— tr_data, val_data, te_data, tr_inds, val_inds, te_inds—are returned for further training.

The ***train_gnn*** function is called within the main function. The configuration stores all model parameters. The edges and nodes are modified as needed (e.g., indices are added) before calling the actual ***train_homo*** function. The data is converted into loaders by the ***get_loaders()*** function, and the ***LinkedNeighborLoader()*** function batches the data.

The model is instantiated using the ***get_model()*** function and is returned from this function. The actual model is imported from the ***models.py*** file, which initializes all predefined models from the ***torch_geometric*** module. The function then calls ***train_homo()***, which trains the model in batches. For each iteration (epoch), it logs the F1 score for the training, testing, and validation data.

The commented code for easy understandability can be found at: *AML_MULTI_GNN*

References:

Provably Powerful Graph Neural Networks for Directed Multigraphs

Realistic Synthetic Financial Transactions for Anti-Money Laundering Models

Contributors:

Harshit Bhushan (Intern NPCI 2024)

Asit Desai (Intern NPCI 2024

Keyur Doshi (Intern NPCI 2024)