Introduction to Deep Learning - Assignment 1

Question 1

Subpart 1.1

Graph 1 has the following properties on the connectivity front:

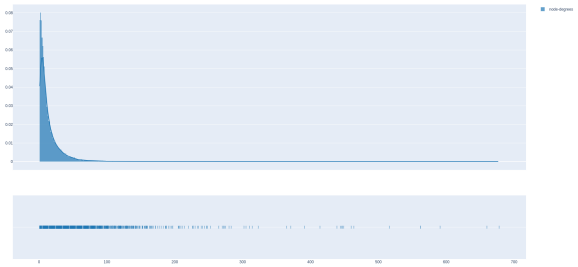
Number of nodes	27917
Number of edges	206259
Max Degree	678
Average Degree	14.776587742235913
Graph Density	0.000529323246247167
Graph Sparsity	0.9994706767537528

Graph 2 has the following properties on the connectivity front:

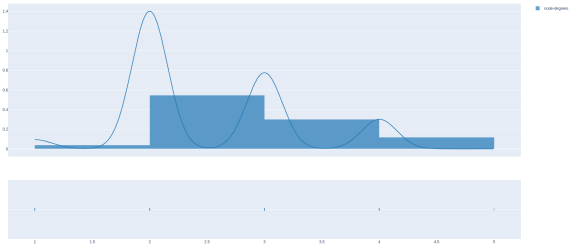
Number of nodes	2642
Number of edges	3303
Max Degree	5
Min Degree	1
Average Degree	2.5003785011355033
Graph Density	0.0009467544495022731
Graph Sparsity	0.9990532455504977

The degree profile of Graph 1 is as follows. The graph degrees show an exponential fall off. The modal degree is 2.

The degree profile of Graph 2 is as follows. The nodes are of degree between 1 and 5 with 2 being the most



common degree, following an almost possionian distribution.



Subpart 1.2

Cliques in the Graphs

For Graph 1:

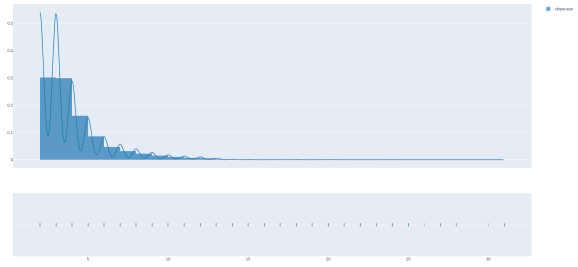
- Number of 3-cliques 387444
- Number of 4-cliques 818754

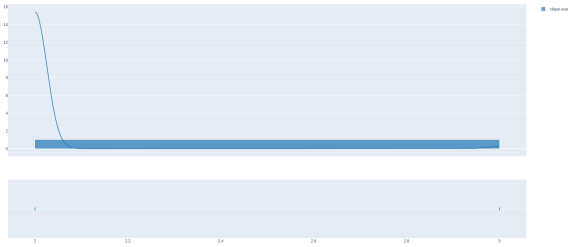
For Graph 2:

- Number of 3-cliques 53
- Number of 4-cliques 0

The distribution of these clique sizes falls of exponentially in the first graph.

There are very very few cliques, upto a maximum of 3 cliques in the second graph.

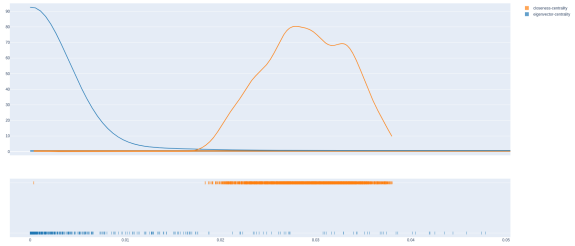




Node Centrality

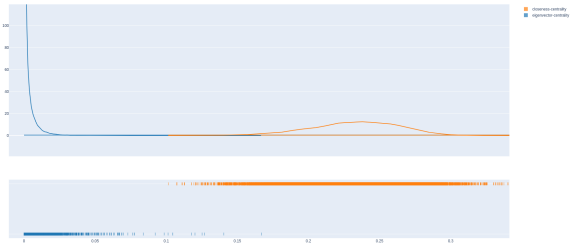
For Graph 1, the centrality figures go as follows:

Mean Eigenvector Centrality	0.00235518780423514
Median Eigenvector Centrality	0.00048050355860991344
Max Eigenvector Centrality	0.16697966868110678
Min Eigenvector Centrality	1.5868796397316522e-13
Mean Closeness Centrality	0.2325952822050173
Median Closeness Centrality	0.23460400699206668
Max Closeness Centrality	0.3483012888495178
Min Closeness Centrality	0.10146402454094762



For Graph 2, the centrality figures go as follows:

Mean Eigenvector Centrality	0.0035310577842978356
Median Eigenvector Centrality	2.4381282767542825e-08
Max Eigenvector Centrality	0.27176584095447287
Min Eigenvector Centrality	-2.039774046532135e-17
Mean Closeness Centrality	0.028929107322656154
Median Closeness Centrality	0.029004014787812137
Max Closeness Centrality	0.038085493935178315
Min Closeness Centrality	0.00037864445285876



Clustering Coefficients

For Graph 1:

Mean Clustering Coefficient	0.2953907581829884
Median Clustering Coefficient	0.20921985815602837
Min-Max range of clutering coefficient	0 to 1

For Graph 2:

Mean Clustering Coefficient	0.015960131213726973
Median Clustering Coefficient	0.0
Min-Max range of clutering coefficient	0 to 1

Final Remarks

The first graph is a lot more clustered, has many cliques of varying sizes, and is very reminiscent of a social structure graph built from some online website's userbase, citation network or the such. This graph is sparse, but the clusters have an exponential size distribution of upto about 600 sized cliques leading us to strengthen the social media hypothesis.

The second graph has very few clusters and cliques, it's almost entirely chains, which leads us to believe that this is obtained from some network of roads where many points on each road were sampled and closest points are joined or the such. This graph is very sparse as well.

Subpart 1.3

We shall take a look at the number of bridge edges to understand if the graph is filamentous or has some cyclic connect when it's sparse.

For Graph 1, it has 2362 bridge edges out of the total 206259 edges, with a bridge percentage of 1.14%, which suggests that this is not a social media graph and instead is some citation network, because even though 1.14% is low it's still higher than bridge values for people in a community.

For Graph 2, it has 141 bridge edges out of the total 3303 edges, with a bridge percentage of 4.2%, which leads us to believe some cyclic structure or the like exists, despite the filamentous nature which leads to a 4.2% bridge edge factor.

Question 2

Subpart 2.1

- Eigenvector Centrality: This measures the steady state probabilities of random walks on the graph. This is the most useful in **indentifying social media influencers on friend graphs**. The reason is that even with a smaller audience directly connected to them, and they friends list being densely connected for other users, the high eigenvector centrality nodes will be the ones who have the **most attention from the most influential people, in some sort of a recursive definition**. Therefore they hold the most social capital and hence are appropriate marketing targets to infuse a product into a community. This is a great heuristic for **graph level classification task**, where the most important webpages or nodes in a networks get high values, SAGEConv, or GATConv for graph classification is an apt example of this.
- Betweenness Centrality: This measures how many shortest paths between two nodes pass through a given node. One useful distinction of betweenness from eigenvector or closeness in real life is that in the traffic map of several highly connected cities, the city capitals or centers will have very low betweenness centrality, but the bridges which connect the cities will have high betweenness. One use of such a centrality is in **making recommender systems for video, literature or scientific literature reading**, where the highest betweenness papers might be **review papers or those that cite crucial papers to prove a result that is then used as an abstractions, and are thereby essential to making connect and learning a valuable abstraction quickly**. This is a good guide for **Edge Level classification tasks**, where the nodes in between are connected to highly visited edges, EdgeConv is a good example.
- Closeness Centrality: This makes that node central which is closest to other nodes, minimizing the sum of shortest path distances to it from all other nodes. **In the graph of a objects of value in a city, placement of fire hydrants and police and fire vehicles** is well guided by such a centrality measure. This is true because we just want to **place this support as centrally as possible to minimize travel time** assuming that traffic is not a major issue. This is great for **Node Level classification tasks,** Message Passing Neural Networks like GCNConv, etc. works well.

Subpart 2.2

The method of WL kernels with color refinement can be used to detect graph isomorphisms. Following are two runs of this algorithm employed on a pair of isomorphic and non-isomorphic graphs.

The idea is that we define a kernel function on our graph which counts the number of nodes of each color.

$$\phi(g) = \left[\frac{\#\text{ nodes with color i}}{k} \;\; \forall i\right] \text{with k such that } ||\phi(g)|| = 1$$

While this is an infinitely sized vector, the dot product of two such vectors is easily computable since the number of non-zero entries is small and ennumeratable. We normalize the vectors, and claim that this kernel is invariant to permutation of nodes, and the probability of two graphs sharing this color vector is small. It gets even smaller when computed over several runs of color refinement.

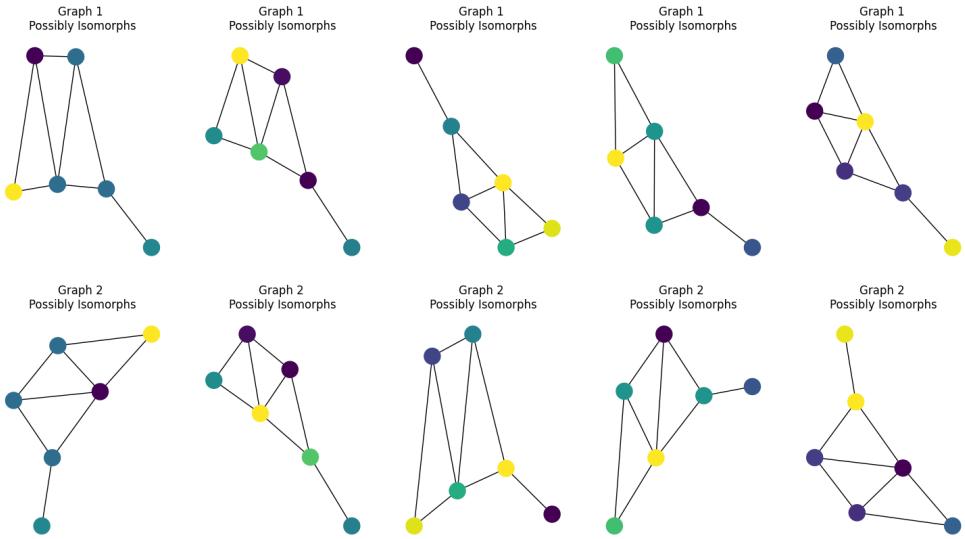
If the graphs are isomorphic, then the inner product in kernel space should be 1.

$$k(g_1, g_1) = \phi(g_1) \cdot \phi(g_1) = ||\phi(g_1)|| = 1$$

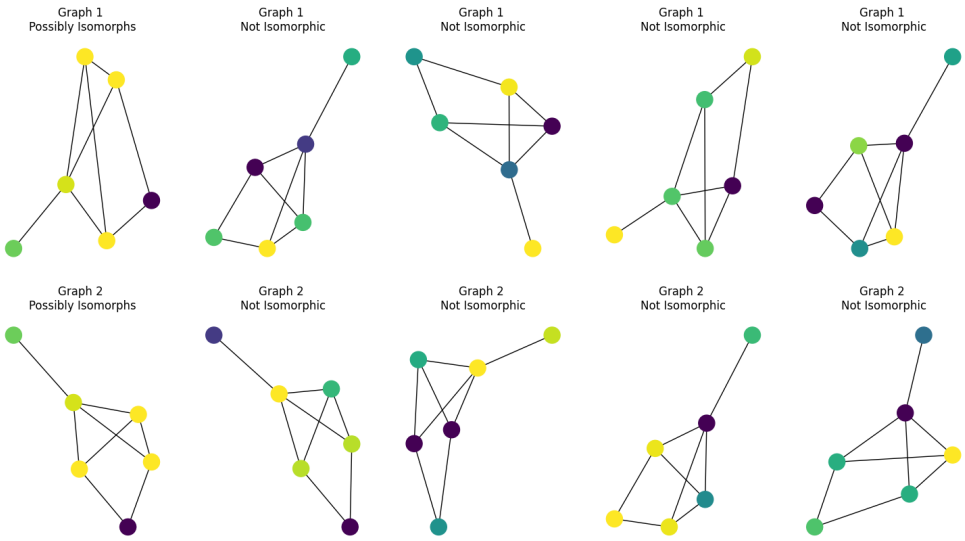
We assign colors based on the degree of nodes, initially. Then in each iteration we update the colors based on the node and it's neighbors colors, using a random hash function which maps evenly back to the color space.

$$c^{t+1}(v_i) = \text{hash}\left(\left[c^t(v_i);\; c^t(v_j) \;\; \forall j \in g_{adj}(i)\right]\right)$$

Using this colors kernel, we get exponentially low probability that two different graphs share the same kernel function, and therefore we if they do, we are extremely confident that they are isomorphisms of each other.



Graphs which are isomorphic to each other show the same color counts over all the iterations of the algorithm.



Non-Isomorphic graphs have different color counts, here as early as in iteration 2. In iteration 1 the equality of node degrees let's us believe that they might be isomorphic, but the second degree features help resolve this.

Code for this algorithm is listed below

```
import numpy as np
import networkx as nx
from matplotlib import pyplot as plt


def check_isomorphism(g_1, g_2, iterations=5):
    """
    Color refinement process on WL kernels to check if two graphs are isomorphic
    """
    color_map_1 = np.zeros(shape=g_1.number_of_nodes(), dtype=np.int32)
    color_map_2 = np.zeros(shape=g_2.number_of_nodes(), dtype=np.int32)
    not_isomorphic = False
    plt.figure(figsize=(25, 10))

    for iteration in range(iterations):
        neighbors_1 = [
            np.sort([color_map_1[neighbor] for neighbor in list(g_1.neighbors(node))])
            for node in g_1.nodes()
        ]
        color_map_1 = np.array(
            [hash(",".join(list(map(str, neighbor)))) % 100 for neighbor in neighbors_1]
        )

        neighbors_2 = [
            np.sort([color_map_2[neighbor] for neighbor in list(g_2.neighbors(node))])
            for node in g_2.nodes()
        ]
        color_map_2 = np.array(
            [hash(",".join(list(map(str, neighbor)))) % 100 for neighbor in neighbors_2]
        )

        if not np.all(np.sort(color_map_1) == np.sort(color_map_2)):
            print(color_map_1)
            print(color_map_2)
            not_isomorphic = True

        plt.subplot(2, iterations, iteration + 1)
        nx.draw(g_2, node_color=color_map_1)
        plt.title(
            "Graph 1 \n"
            + ("Not Isomorphic" if not_isomorphic else "Possibly Isomorphs")
        )
        plt.subplot(2, iterations, iteration + iterations + 1)
        nx.draw(g_2, node_color=color_map_2)
        plt.title(
            "Graph 2 \n"
            + ("Not Isomorphic" if not_isomorphic else "Possibly Isomorphs")
        )

    plt.show()
```

```
    return not not_isomorphic


if __name__ == "__main__":
    g_1 = nx.Graph()
    g_1.add_nodes_from([0, 1, 2, 3, 4, 5])
    g_1.add_edges_from(
        [
            (0, 2),
            (0, 4),
            (0, 5),
            (1, 4),
            (1, 5),
            (2, 3),
            (2, 4),
            (4, 5),
        ]
    )

    g_2 = nx.Graph()
    g_2.add_nodes_from([0, 1, 2, 3, 4, 5])
    g_2.add_edges_from(
        [
            (5, 0),
            (5, 2),
            (5, 4),
            (1, 2),
            (1, 4),
            (0, 3),
            (0, 2),
            (2, 4),
        ]
    )
    check_isomorphism(g_1, g_2)

    g_3 = nx.Graph()
    g_3.add_nodes_from([0, 1, 2, 3, 4, 5])
    g_3.add_edges_from(
        [
            (0, 2),
            (0, 4),
            (0, 5),
            (1, 5),
            (2, 4),
            (4, 5),
            (4, 3),
            (1, 2),
        ]
    )
    check_isomorphism(g_1, g_3)
```