Università della Svizzera italiana

Facoltà di scienze informatiche

Graph Deep Learning **SP 2021**

Student: Maria Andromachi Kolyvaki

Project: Clique Pooling for Graph Classification

1. Introduction

With the development of Graph Networks different deep learning models have been applied to irregular graphs. Undoubtedly pooling is a key component of traditional CNNs, which enables a non-linear downsampling. Apart from space reduction purposes, pooling layers also used for the extraction of dominant features of highly regular graphs, as images being in nature. However, GNN methods are considered to be flat and that could be categorized as a problematic in the task of Graph classification [2]. In the paper *Clique Pooling for Graph classification* [1], there has been proposed a new pooling layer for Graph Networks, that introduces a particular type of relational inductive bias, which is that of hierarchy processing. The advantages of considering a pooling layer that is in nature static and non parametric can be illustrated by both the ease of its implementation, but most importantly by its performance when compared with state-of-the-art approaches in pooling for well known datasets on MUTAG for Graph Classification. By considering maximal cliques, the aforementioned pooling layer achieves similar objectives as other pooling layers, like PatchySan on Proteins dataset and Set2Set when tested on DD dataset.

2. Topology of Cliques

The word cliques is well known in graph theoretic literature and also has been widely used in circuit design. As we know, the problem of finding maximal cliques, is an NP-complete problem and a well known algorithm for computing the maximal cliques has be given by Bron–Kerbosch which provides an enumeration them, while having a runtime complexity of exponential order. This algorithm is the one used in our implementation of the pooling layer and it is also provided from NetworkX library. Let's firstly go into to two definitions that will be useful for a better understanding of the implementation.

Cliques

Given an undirected graph $\mathbb{G} = (\mathbb{V}, \mathbb{E})$, a clique is a subgraph which forms a complete graph.

Maximal Cliques

Are the cliques that cannot be extended by including one or more adjacent vertices. In other words, maximal does not refer to the size of a clique, but it refers to the fact that is not further extendable.

Undoubtedly, the maximal cliques could be considered as a promising candidate for simulating neighborhood in graphs, as the nodes that are contained in the maximal cliques are strongly connected with each other. So it could be theoretically and ,as we will see later, also practically a good way to simulate the spatial locality that is an established fact in standard CNN architectures. Highly irregular graphs are hard to manipulate let alone to draw inferences about their complex topological structure. In our pursuit of trying to find good key identifiers that represent spatial locality, maximal cliques could be considered as valuable candidates.

3. Implementation

We would like to start the description of our implementation of the Bron–Kerbosch algorithm. Initially, we implemented ourselves the algorithm. However, in order for the layer to be more diverse and to use as many built-in methods of other libraries as possible, that are vastly used by other developers, later was decided to use the algorithm that was already implemented in the NetworkX library. For the record, we will provide a snippet of the algorithm, since it is not enclosed in python notebook and additionally is particularly brief.

```
def Bron_Kerbosch(potential_clique=[], remaining_nodes=[], skip_nodes=[]): # namely R, P, X

  if len(remaining_nodes) == 0 and len(skip_nodes) == 0:
    print('This is a clique:', potential_clique)
    all_cliques.add(tuple(sorted(potential_clique)))
    return 1

  found_cliques = 0
  for node in remaining_nodes:

    new_potential_clique = potential_clique + [node] # R = R ∪ {v}
    new_remaining_nodes = [n for n in remaining_nodes if n in node.neighbors] # P = P ∩ neighbors(v)
    new_skip_list = [n for n in skip_nodes if n in node.neighbors] # X = X ∩ neighbors(v)
    found_cliques += Bron_Kerbosch(new_potential_clique, new_remaining_nodes, new_skip_list) # call function recursively

    remaining_nodes.remove(node) # P = P \ {v}
    skip_nodes.append(node) # X = X ∪ {v}
  return found_cliques
```

Figure 1. Bron–Kerbosch algorithm

To what comes next, we will briefly analyze the implementation. Firstly, let's start with the method, get_neighbours which as its name suggests computes all the neighbors of each node, namely the nodes that are in the same clique with our current node. This means that in this approach we consider as neighboring nodes all these that belong in the same clique. Next, as we used the implementation of

NetworkX for finding the maximal cliques, method to_networkx implements exactly this conversion from a graph to a NetworkX graphinstance on which we will call the built in method: find_cliques to enumerate them.

For example as you can see in Figure 2, the assignment of nodes to the pools is done in a greedy method ranked by size. This means that a node belonging in parallel to more than one clusters, namely B, will be assigned to the cluster having bigger size in terms of nodes, i.e. 1 (the grey one). However, in case of equally sized cluster (node D) the nodes that are contained in both of these clusters, will finally remain clustered to both of them and in the aggregation procedure of the node features will contribute in the same way for the respective clusters. For that reason we lack the usual definition of clusters that each node should belong to one cluster, so in what follows we will refer to these clusters as fuzzy clusters. This is exactly what is implemented in the get_nodes_clusters method which after sorting the clusters with respect to their length, filters out all these clusters whose nodes have been already processed in previous clusters.

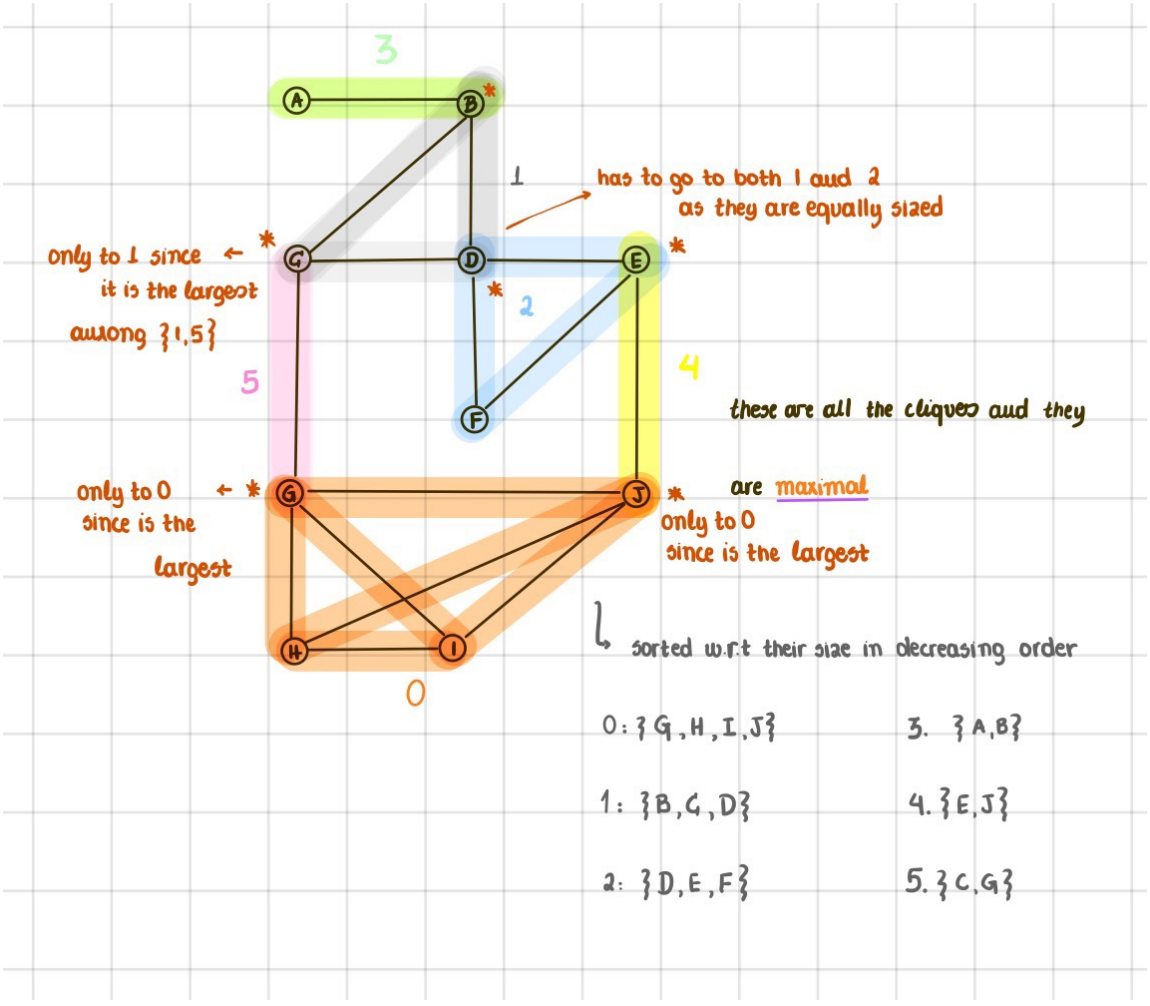

Figure 2. Illustrative example of the implementation

So the considered cliques will be the ones as in Figure 3, since nodes of cliques 4 and 5 have been covered by other cliques they are not considered.

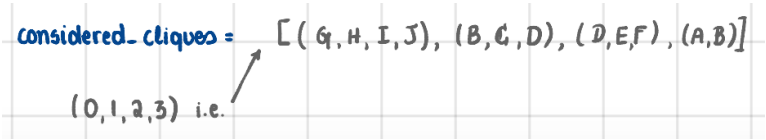

Figure 3

As a next step we would like to get all the intra-cluster edges, in order to maintain the connectivity of the nodes of the initial graph that did not belong to the same cluster, but also for the vertices that were assigned to more than one cliques, as we need them in order to form the edges in the residual graph after the coarsening. The edges that the clique members entail will be inherently moved to the "next generation", meaning the residual graph.
This is implemented in get_dual_edges method after adding the sibling edges as we name them, which in practice are the edges that are created in between the clusters that are equally sized and share a node. To make things more clear, we would like to explain our approach for this step. As we have two fuzzy clusters, we need somehow to account for this intersection in the residual graph and to pass this information on to the next pooling layers, this is done by introducing a new edge, which can be understood as follows. If we replicate the node in the intersection and we add an edge connecting them then we get this sibling edge.
However, in our code no node duplication takes place, but this example was just used as an illustrative example, for the introduction of this new edge. All in all, this edge will be the new intra-cluster edge for the two fuzzy clusters that have common intersection. Then we compute for each cluster its siblings, namely all these clusters that shares an edge with, and in order to find them we subtract from all the neighbors of the nodes contained in the cluster those that are internal in the cluster. Last but not least, in forward method we compute the dual node features and the dual edges. Here duality refers to the reduced representation of the initial graph, along with the new super-node features that were a result of the inter-cluster node feature aggregation.
Lastly as described in the paper, in order to coarsen the node features we aggregate the maximal cliques. As specified in the paper, as a readout function that is in favor of coarsening the node features we use either average or max function over the nodes that are within the same pool.

4. Performance based on paper benchmarks

We compared the performance of our model with respect to the Proteins dataset, and as you may see in the attached python notebook, we have achieved an overall accuraccy of 72.2% which is almost as good as the one provided in the paper.
We also proceeded in some checks for the implementation of the layer itself, when applied to the graph of Figure 2, and after the application of the clique pooling both the clusters and the intra-cluster edges where computed as expected.

Lastly, we performed an ablation study in order to study what were the contributions of the Clique Pooling layer. All in all, the overall accuracy was increased by more or lesst 6%, and in general provided more stabilized results in contrast when no pooling layer was used in the model.
Apparently the paper itself was not deeply descritpive about the exact model they used and the combination of hyperparameters they used, so some of the benchmark behaviour was difficult to be implemented. In addition, we found no online resources from their paper regarding any kind of testing they implemented for the paper.

5. Conclusion

In the final analysis, clique pooling being in nature static and non parametric, could be easily interpreted and draw conclusions about it. Having such a straightforward implementation and not necessitate problem-specific treatment, makes it universal and easily used. Even if it does not achieve the state-of-the-art for the datasets it is tested for, provides a promising solution for pooling in Graph Networks. All the aforementioned, along with the results that were provided in the original paper when being tested as a dupe of pooling in traditional CNNs, without being too far in terms of accuracy with the best pooling techniques used in these neural networks.

References

[1] Luzhnica, E., Day, B., Lio, P. (2019). Clique pooling for graph classification. arXiv preprint arXiv:1904.00374.

[2] Ying, R., You, J., Morris, C., Ren, X., Hamilton, W. L., Leskovec, J. (2018). Hierarchical graph representation learning with differentiable pooling. arXiv preprint arXiv:1806.08804.

[3] PyTorch Geometric (PyG), https://github.com/rusty1s/pytorch_geometric

[4] GNNs and related works list, https://github.com/hazdzz/GNNs_and_related_works_list