Machine Learning with Graphs: Homework I - Graph Algorithms Due: 01/31

Spring 2023

Reminder: You can choose to solve problem 2 or 3.

Problem 1

In the binary node classification problem, the goal is, given a single graph $G = (V, E)$ and a set of train $T \subseteq V$ and test $S \subseteq V$ vertices, $S \cap T = \emptyset$, to learn a function $f(v) \approx y_v$, where $y_v \in \{0, 1\}$, based on V'. We will consider a very simple function f that, for a given value of k, returns the most frequent label for nodes within k hops from $v \in S$—unlabeled k-hop neighbors are ignored. Initially, assume that we don't have any labeled nodes but are given a set of candidates $C \subseteq V \setminus S$ from which we can pick r nodes to be labeled (T).

a. Based on our classification model, we are unable to predict the labels for nodes that don't have at least one labeled k-hop neighbor. Thus, our goal is to select r neighbors that will maximize the number of predictions. The associated decision problem is whether labels for h nodes can be predicted with r labeled ones. Either show that the decision problem is NP-complete or propose a poly-time algorithm. *Hint: vertex cover problem.*

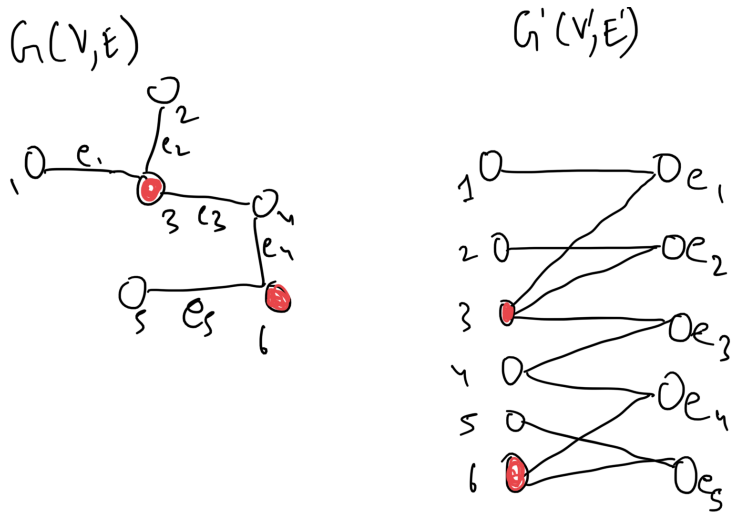
b. Consider the following heuristic for the previous problem. Start selecting $v_1 \in C$ with the most k-hop neighbors in S. Next, select the node $v_2 \in C$ with the most k-hop neighbors in S that are not already *covered* by v_1. Repeat until r nodes are selected. Assuming an efficient implementation of this heuristic, what is its complexity? Is it optimal? If not, try to come up with an example where it performs poorly (i.e. not optimal).

Ans a: I will try to prove that the k-HOP problem is NP-complete. The first step is to prove that given a solution to the K-HOP problem, we can verify it in poly-time, this will ensure that K-HOP is in NP.

If we are given the solution to K-HOP, then we can do breadth first search starting from the nodes in the set 'R'. If all the test nodes appear during the BFS, then the solution is correct otherwise it is not. The time complexity for BFS is O(V+E) which is poly-time in the number of nodes.

Now to prove that K-HOP is NP-complete, i need to reduce a problem which is NP-complete to this K-HOP problem. I will reduce vertex cover to K-HOP as follows:

Let G(V,E) be a graph, then for each edge(u,v) in G, create a new node 'e'. Connect the new node 'e' to both the nodes of the edge (u,v), repeat this for all the edges in graph G. The newly constructed graph is G'(V',E'). The nodes in G are the nodes in the train set, the newly added nodes in G' are the nodes in the test set. An example is shown below, this example is when I consider the case of K-HOP =1, it can easily be generalised for any value of K-HOP by simply adding K nodes when constructing the graph in a similar manner.



Now, i have to prove that if I can find a vertex cover in G, then i have solved the K-HOP (1-HOP in this case) problem.

Proof (Vertex Cover → K-HOP)
If there exists a vertex cover in G, lets say the size of vertex cover is 'r', then the corresponding nodes solve the K-HOP (1-HOP in this case) problem in G'. This can easily be seen in the above picture, the red nodes are the vertex cover and they also solve the 1-HOP problem in G'. The conversion from G to G' is poly-time so the reduction is valid.
Proof (K-HOP → Vertex Cover)
If I can get the solution to the K-HOP problem, then I can easily get the vertex cover, i.e if i know which 'r' nodes are needed for the k-hop problem in G', then

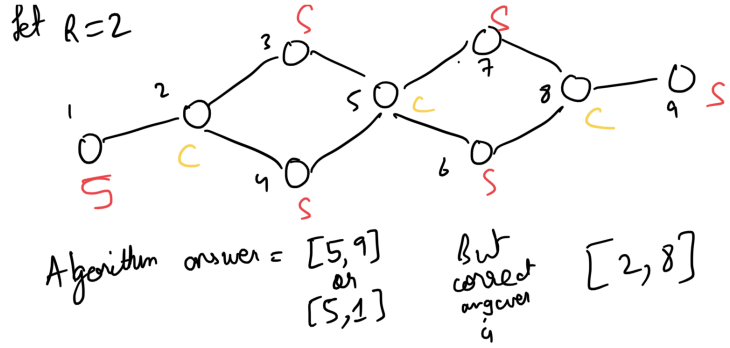
those same 'r' nodes will be the vertex cover in G. It can be easily seen from the above image as well. This reduction is also poly-time and hence valid.

I showed the above reduction for K-hop=1, it easily be generalised for any K-hop. Since i reduced the vertex cover to K-hop problem for K=1 which is a subset of the K-hop problem, i have proved that K-HOP problem is NP-complete.

Ans b: Analysing the time complexity of the mentioned algorithm: Since we are selecting V1 which has the most K-HOP neighbors in S, we will have to do a BFS on each node in the candidate set and then determine the maximum. This operation will take C*O(V+E) time, here i have mentioned O(V+E) for the BFS, 'C' is for the number of nodes in the candidate set. O(V+E) is actually an over approximation because we will just do BFS of K steps, but O(V+E) is the worst case. This is the amount of time for selecting V1. Next for V2, the same process will be repeated but time complexity will be (C-1)*O(V+E) as one node from the candidate set has already been chosen. This will be repeated until we select 'R' nodes.

T(N)¡=$\Sigma_{r=0}^{r=r}(C-r)*K(V+E)$, here 'K' is just a constant as i removed the big O notation. It can easily be seen that is algorithm runs in poly-time.

The solution provided is a greedy solution, it is not optimal, it fails at some points thats why K-HOP is a NP-complete problem and not a part of P. One case where it fails is shown below:



Using the algorithm, we can get any of the three answers shown above, but the correct answer is only one of them.

Problem 3 (optional)

You will implement and evaluate a subgraph-based similarity function for labeled graphs. For each graph, count the frequency of occurrences for every possible connected subgraph with k vertices. As an example, $k = 1$ you will get the frequency of vertices with each label. For $k = 2$, you will get the frequency of edges with a given combination of vertex and edge labels. For $k = 3$, you

will get the frequency of every (connected) labeled triad. A graph will be represented by a vector with the frequency of each subgraph. Then the similarity between the two graphs can be computed using the inner product between their respective subgraph frequency vectors. This is called a *graphlet kernel*.

a. Implement the proposed method in Python assuming that both vertices and edges are labeled. You can use the networkx library[1] to manipulate the data.

b. Use the MUTAG[2] dataset to evaluate the running time of your implementation for different values of k.

c. Use the MUTAG graph labels to compare the similarity between graphs in the same and in different classes—you can use your favorite set of statistics (mean, standard deviation, distribution, etc.).

Ans: Link to code

[1] `https://networkx.org`

[2] `https://chrsmrrs.github.io/datasets/docs/datasets/`