Graph Colouring: An ML based approach

Amey Kulkarni

23rd November

0.1 Introduction

The problem of graph colouring, or more specifically, vertex colouring of a graph is a well known NP complete problem. It is also extremely relevant since other graph colouring problems (edge colouring, face colouring, etc) can be converted to it. Graph colouring is a relatively popular problem even in the general public domain through sudoku puzzles and map colouring.

0.2 Formal Problem Definition

Given an undirected graph G, find the minimum number of different colours required to colour the graph subject to the following conditions-

- Each vertex is coloured using exactly one colour.
- No two adjacent vertices (two vertices joined by an edge) share the same colour.

0.3 Related Work

The method used is to create the positive and negative training examples, along with the two graph operations defined in the "Approach" section.

Refer Meyer's paper and "Deep Learning meets Graph Colouring".

0.4 Approach

0.4.1 Summary

The algorithm used can be broken down into the following steps:

- Generate k-partite graphs with specified density. Create a train-test split.
- During training-
  - Created a feature vector for each node of the graph based on its "neighbourhood characteristics".
  - Sample pairs of non-adjacent points. Obtain their feature vectors and concatenate them to form the training data. Obtain the training labels from whether or not they belong to the same partition.
  - Perform one of the two possible operations on the nodes based on the label obtained.
- Feed the training data to a classifier. I used *sklearn's Linear Regression.*
- During testing/prediction-
  - Create feature vectors and sample points randomly in the same way.
  - Use the classifier to predict which of the two operations to perform.
  - Repeat the procedure until a clique is formed. Each vertex of the clique can be thought to have a unique colour. All vertices merged inside each of the final vertices have the same colour as that of their final vertex.

0.4.2 Operations

The two following operations are defined on a graph G for any two non-adjacent vertices u and v-

- Merge: The two nodes u and v are merged into one node, say w. Edges connecting either of the vertices, that is, all edges of the form $u - x$ and $v - x$ for a given node $x \in V(G) - \{u, v\}$ are now edges of the form $w - x$, where $V(G)$ represents the vertex set of graph G.
- Add an Edge: A new edge $e = \{u, v\}$ is added between the two vertices.

The logic behind it is that merging is the same as colouring the vertices with the same colour. They become one single vertex. Adding an edge on the other hand necessitates that these two vertices be coloured different, since no two adjacent vertices can have the same colour according to the problem.

0.4.3 Feature Vector

The training samples consisted of labels that decided whether the merge operation or add-an-edge operation was the right operation to perform at any given stage. This decision had to be made based on the two vertices u and v in question and also their neighbourhood properties. Hence, a method was required to accurately capture such neighbourhood properties. Moreover, it had to be fast enough, since the graph was ever-evolving. The following are the different methods I used for generating feature vectors for nodes-

1. Manual feature vectors- For every node, find out its neighbourhood characteristics such as number of neighbours, whether it is a leaf or not, etc. This method gave poor results since it did not represent all the information required for the sampling process to work. It was not extremely efficient either since at each step the graph changed, it required $O(N)$ time to update the feature vector.
2. Top k eigenvectors- Extract the top k eigenvectors of the adjacency matrix of G. Stack these eigenvectors columnwise to obtain an $n \times k$ matrix. Hence, for every node, we get a k sized feature vector.
3. Node2Vec: Algorithmic framework provided by Stanford University that produces feature vectors of specified dimensions. It is similar to the Word2Vec algorithm in that they both use positive and negative samples as training examples. Node2Vec does this by generating walks of specified lengths to find the neighbourhood characteristics of nodes and subsequently the "context" for each node.

Optimisation over accuracy- Since the graph constantly updates and changes after every operation (merge and add-edge), it is very time consuming to always recalculate the feature vectors after each operations. In fact, the step of calculating the feature vector is the most intensive step in the entire algorithm. A simple trick to reduce this time is to set an *update-interval.* Only after every *update-interval* operations will we recalculate the feature vectors. The reasoning behind this is that the neighbourhood characteristics of the graph (and as a result the feature vectors) will not drastically change with a few (5 or so) operations, and the overall probability of choosing a vertex that has just been part of a merge operation is quite small. This trick reduces the time required by a factor of *update-interval*.

0.5 Dataset

It was important to generate the dataset in such a way that it would be possible to find the correct operation at each step. Hence, the dataset was always created as a k-partite graph with at least one k-sized clique. This ensured that the optimal graph colouring always consisted of exactly k colours. More importantly it made sure that the optimal step to perform was a *merge* if and only if vertices belonged to the same partition and *add-edge* if and only if they belonged to different partitions. Hence, the labels were always accurate.

The rest of the edges between vertices are added randomly, with a probability p. p can be varied to form dense or sparse k-partite graphs.

Varying p, the number of partitions and the number of nodes in each partition leads to sufficiently random graphs. At present all partitions have the same number of nodes, but this can be changed in the future.

0.6 Results

0.7 Future Work

0.8 Rough Points

May help to simply make the graphs easier, as an intermediate step.