Midterm Review: Graph Neural Networks for Minimum Spanning Trees

Brandon Kates
Cornell University
bjk224@cornell.edu

Michael Lapolla
Cornell University
mel259@cornell.edu

Peter Haddad
Cornell University
ph387@cornell.edu

Abstract

Deep Neural Networks are ubiquitous for solving Natural Language Processing and Computer Vision tasks, and are revered as being able to solve any problem given the right model. There has been a clash between traditional optimization algorithms, which have rigorous mathematical proofs showing how they work, and deep learning 'black box' algorithms which work extremely well but which are much more difficult to understand. We want to evaluate the ability of deep learning models on more traditional optimization problems, like finding the minimum spanning tree of a graph, to see how they perform given some constraints.

1. Introduction

We want to understand the limitations of deep learning (hereafter referred to as DL). We intend to quantify how difficult it would be to solve very simple combinatorial problems with architectures intended for more complex problems. We want to continue on prior work that has been done in the area to see where we can push it. Graph Neural Networks (hereafter referred to as GNN's) have been applied to the traveling salesman problem (TSP) and Dijkstra's, but none that we found specifically for the minimum spanning tree algorithm.

A minimum spanning tree is a subset of the edges of a connected, edge-weighted undirected graph that connects all the graph's nodes together with the minimum possible total edge weight. That is, it is a spanning tree whose sum of edge weights is as small as possible. A spanning tree is a subset of the edges of a connected graph that connects all nodes but does not have any cycles (some number of nodes (at least 3) connected in a closed chain of edges).

It is our goal to build a Graph Neural Network architecture that can take in any graph (defined as G=(V,E)) and solve the minimum spanning tree algorithm. In its current state, our GNN takes in fully connected Euclidean graphs and attempts to produce its minimum spanning tree. We eventuality want to see how our architecture compares to Prim's and Kruskal's algorithm for solving minimum spanning trees, given a fixed amount of training time, a fixed number of examples, or other limited resources.

Firstly, we need to solve the problem of generating random graphs with their corresponding minimum spanning tree solutions that we can then feed into our network. Once we are able to generate random graphs of any size and produce a solution, as part of our end goal we want to fully evaluate the performance of our model architecture compared to the ground truth minimum spanning tree solution. We would like to observe accuracy of the model, how close it is to the actual minimum spanning tree and how often (if ever) it fails to produce a valid spanning tree. Additionally, we wish to analyze how the model learns, if it follows a similar structure to the combinatorial methods mentioned earlier, whether it comes up with its own way of learning, or if it is not able to learn how to produce a minimum spanning tree solution.

2. Architecture

We decided to build on the work of Chaitanya K. Joshi, Thomas Laurent and Xavier Bresson in their paper *An Efficient Graph Convolutional Network Technique for the Travelling Salesman Problem*[1], where they use deep Graph Convolutional Networks to build efficient TSP graph representations and output tours in a non-autoregressive manner via highly parallelized beam search. Their approach outperforms all recently proposed autoregressive deep learning techniques in terms of solution quality, inference speed and sample efficiency for problem instances of fixed graph sizes. And thus we deemed it to be a perfect base for us to build our alterations and architecture on top of.

Joshi, Laurent and Bresson used supervised learning to construct feasible traveling salesman tours on an input training dataset of 1,000,000 fully connected Euclidean graphs, each with either 20, 50 or 100 nodes, and their corresponding TSP solution.

In order to use this architecture in our work, we needed to alter the existing code related to the Traveling Sales-

[1]https://arxiv.org/pdf/1906.01227.pdf, https://arxiv.org/abs/1906.01227

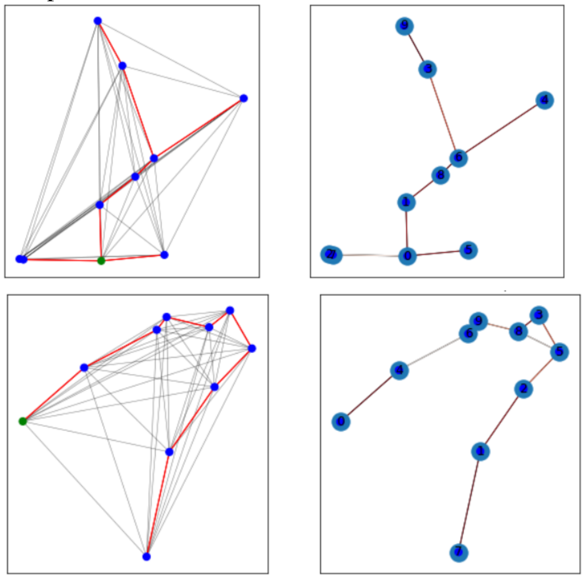
man Problem to better align with our desired objective - the Minimum Spanning Tree problem. This involved making large changes to the existing DL system. One of the largest changes was creating a different search method to predict what edges belong in our MST prediction- rather than the existing search method to predict what belonged in the TSP prediction. To construct a valid TSP tour, they used a version of Beam Search[2] to determine the most promising edges to include. Whereas we created and are using a probability based greedy heuristic.

3. Graph Generation

The first stage of our solution was to build a system to solve Euclidean minimum spanning tree problem (the MST on only Euclidean graphs) and then generalize this system to work on all graphs (Euclidean and non-Euclidean). We generated approximately 1,000,000 fully connected Euclidean graphs of size 10 nodes and their minimum spanning tree respectively to train our model. Next we generated 10,000 graphs and their corresponding minimum spanning trees for testing and validation. To generate the graphs we used Python's NetworkX[3] library specifically the "random geometric graph generator" which places "n" nodes uniformly at random in the unit square and creates edges from every single node to all other nodes.

4. Current Results

In these two examples, the figure on the left shows the original graph with the correct MST in red. The image on the right shows what our GNN predicts the MST would be. Currently, on a test set of size 10,000, 14.14% of the predicted MSTs were identical to the true MSTs.



[2]https://www.ijcai.org/Proceedings/05/Papers/0596.pdf

[3]https://networkx.github.io

5. Next Steps

We will continue to update and improve our search methodology to increase our GNN's accuracy in predicting the MST solutions to Euclidean graphs. Once we are satisfied with that, we will generalize our model to find the MST of non-Euclidean graphs, then we will train test and validate our GNN on graphs generated by various different methods. (such as Erdős–Rényi[4] and Wattz-Strogatz[5]).

[4]http://www.cs.cornell.edu/courses/cs485/2006sp/lecture%20notes/lecture1.pdf

[5]https://pdfs.semanticscholar.org/13e6/f40333f9bbb06634fab7a04c5a9f826971b4.pdf