Graph Neural Networks for Minimum Spanning Trees

Brandon Kates* [1] **Peter Haddad*** [1] **Michael Lapolla*** [1]

Abstract

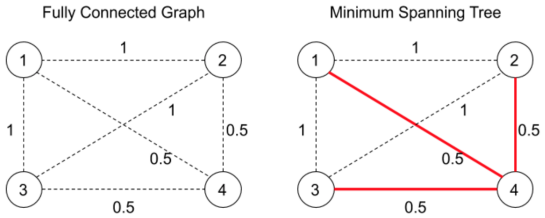
Every so often there is a news headline about machines surpassing the top ability of a human. Examples range from IBM's Deep Blue beating chess grandmaster Gary Kasparov[1], a machine learning system outperforming the top doctors in identifying cell mutations[2], IBM's Watson winning a game of Jeopardy![3] on live tv (and Watson beating the house of representatives in an unrecorded game of Jeopardy!) and most recently, Google's deep mind beating the top professional players in the game Go.[4] While these feats of machine outperforming man are impressive and provide great illustrations for how far the field of machine learning/AI has come-we wanted to see how well a deep learning system could perform on a problem whose solution is already known and simple to implement.

1. Introduction

We sought to understand how a deep learning (hereafter referred to as DL) system would perform on very simple combinatorial optimization problems. We wanted to continue on prior work that had been done in the area to see where we could push it. There is published literature showing the benefits of Graph Neural Networks being applied to the traveling salesman problem (TSP) and the Shortest Path Problem but we found no examples of a Graph Neural Network being applied to the Minimum Spanning Tree Problem.

The minimum spanning tree (MST) problem is a combinatorial optimization problem where the input is a weighted connected graph and goal/output is a subset of edges from the original graph that connect all nodes with the minimum cost.

Figure 1. Example of MST



Further explanation of a spanning tree and the minimum spanning tree of a graph: 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).

We decided to use the Minimum Spanning Tree Problem because it is a conceptually easy problem to understand and there are many known existing algorithms that can solve it efficiently. Some of the prominent algorithms that can solve the MST problem are Prim's, Kruskal's and Borůvka's algorithm. If implemented properly, the MST can be solved in $O(|E|log(|V|))$ where $|E|$ is the number of edges and $|V|$ is the number of nodes in the graph. We specifically decided to use the Euclidean Minimum Spanning Tree Problem because the existing work/literature that we wanted to expand on used the Euclidean MST problem. The Euclidean MST is the same as the normal MST problem except it only looks at Euclidean graphs (Graphs where the weight of an edge between any two nodes is equal to the distance between the nodes).

*Equal contribution [1]Cornell University, Ithaca, NY, USA. Correspondence to: Brandon Kates <bjk224@cornell.edu>, Michael Lapolla <mel259@cornell.edu>, Peter Haddad <ph387@cornell.edu>.

Proceedings of the 35^{th} International Conference on Machine Learning, Stockholm, Sweden, PMLR 80, 2018.

[1]http://theconversation.com/twenty-years-on-from-deep-blue-vs-kasparov-how-a-chess-match-started-the-big-data-revolution-76882

[2]https://www.ncbi.nlm.nih.gov/pmc/articles/PMC6446069/

[3]https://www.techrepublic.com/article/ibm-watson-the-inside-story-of-how-the-jeopardy-winning-supercomputer-was-born-and-what-it-wants-to-do-next/

[4]https://www.wired.com/2016/01/in-a-huge-breakthrough-googles-ai-beats-a-top-player-at-the-game-of-go/

2. Related Work

The framework we first used to start our research was from (Joshi et al., 2019) who used a non-autoregressive deep learning approach for approximately solving the Traveling Salesman Problem using a Graph Convolutional Neural Network designed from [Bresson and Laurent] (Bresson & Laurent, 2017) and a beamsearch. In (Joshi et al., 2019) they use deep Graph Convolutional Networks to build efficient TSP graph representations and output tours in a 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.

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.

Jie Zhou, Ganqu Cui, Zhengyan Zhang et al explore Graph Neural Networks in their paper Graph Neural Networks: A Review of Methods and Applications (Zhou et al., 2018). They found that recent advances in neural network architectures have enabled successful learning with them graph neural networks. Recently systems based on variants of GNNs such as graph convolutional neural networks, graph attention networks etc. have demonstrated enhanced performances on solving the aforementioned problems. In this paper, they provide a detailed review over existing graph neural network models. We read the aforementioned paper to get up to speed and learn more about graph neural networks in order to familiarize ourselves with the problem we are tackling.

While we started off solving MSTs for fully connected euclidean graphs, we ultimately want to transcend from just being in euclidean space and see if we can calculate the MSTs of non-euclidean graphs as well (for applications other than distance-related problems like resistivity between nodes etc.)

We explored Google DeepMind's OpenSpiel framework as well and deemed it to be very useful. OpenSpiel (Lanctot et al., 2019) is a collection of environments and algorithms for research in general reinforcement learning and search/planning in games. The framework includes over 20 implementations of games of various types (perfect information, simultaneous move, imperfect information, gridworld games, an auction game, and several normal-form / matrix games). Game implementations are in C++ and wrapped in Python. Algorithms are implemented in C++ and/or Python.

For the last two months of our project timeline, we pivoted and focused on implementing and testing a MST game in Google DeepMind's OpenSpiel framework which was detailed above.

3. Dataset

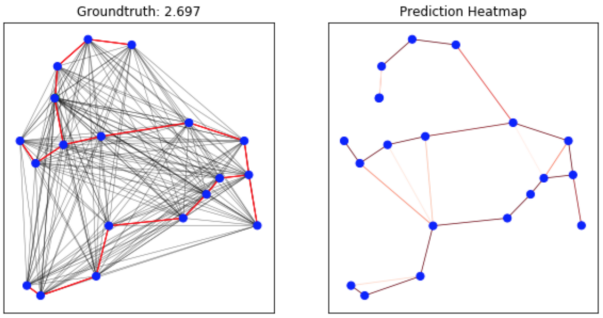
We built a dataset for the 2D Euclidean MST, although the technique can also be applied to sparse graphs. Given an input graph, which consists of n nodes in the unit square, $S = \{x_i\}_{i=1}^{n}$ where each $x_i \in [0, 1]^2$ we first compute the adjacency matrix, A which contains the distance between nodes in the graph represented as edges. We generated 1 million graphs and their solutions for training, and 10,000 graphs for validation and testing of size $n \in 5, 10, 20, 30, 40, 50, 100$ nodes. We used the NetworkX (NetworkX developer team, 2014) (You et al., 2018) library for graph generation and generating minimum spanning tree solutions.

4. Methods

4.1. GCNN + MCTS

Given a graph as input, we train a graph ConvNet model to output a prediction heatmap for each edge in the graph. The prediction heatmap gives the learned probabilities of selecting each edge as part of an MST solution. Using the prediction heatmap, we evaluate the most likely set of edges using a tree-search algorithm known as Monte Carlo Tree Search (MCTS). MCTS returns the final predictions for the edges to include in our predicted MST. Our approach is titled GCNN + MCTS (Graph Convolution Neural Network + Monte Carlo Tree Search).

Figure 2. Groundtruth minimum spanning tree and MST length (left), and the prediction heatmap from the GCNN model (right). The darker the color in the prediction heatmap, the higher probability it should be included in the final predicted MST.



4.2. Graph Convolutional Network

Given a graph as input, the Graph Convolutional Network outputs its prediction heatmap (Figure 2). To get to our GCNN, we built on the work of Chaitanya K. Joshi, Thomas Laurent and Xavier Bresson (Joshi et al., 2019) where they used Graph Convolutional Networks to solve another com-

binatorial problem: the traveling salesman problem (TSP). Their model is a graph ConvNet + beamsearch (Furcy & Koenig, 2005) to find the the shortest path visiting all nodes in a graph.

In order to use this architecture in our work, we needed to alter the existing code related to the Traveling Salesman 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[5] to determine the most promising edges to include. Whereas we created and are using a probability based greedy heuristic.

However, after multiple attempts at trying to generate actual minimum spanning trees, we were unable to completely understand and modify their version of beamsearch to work in a way that generates minimum spanning trees instead of valid traveling salesman tours. The challenge lies in the fact that valid TSP tours are just an ordered set of nodes, whereas MSTs are unordered sets of edges, and the mask within their implementation of beamsearch is coded in a way that only works with nodes.

4.3. OpenSpiel

We implemented the minimum spanning tree as a game in OpenSpiel (Lanctot et al., 2019). Games are represented as a collection of states, actions, and rewards. We define the *state* of an MST game as the currently selected edges in the adjacency matrix. The *actions* or *legal actions* are represented as a list of edges that can be selected in the next time step. The *reward*,r, for taking action, a, during state s, at time t is defined as the negative length of the edge weight (or the distance between two nodes in our graph).

We built the implementation of the MST game by heavily editing OpenSpiel's Tic-Tac-Toe game implementation. Our edits include:

- Changed the number of players from two to one.
- Changed the state to represent the set of chosen edges at each time step.
- Each time step is defined as picking one edge in the n-node graph.
- Changed the starting state to be the adjacency matrix of our graph, detailing current available edges to be chosen.
- Represent the termination state to be states with $n-1$ edges.
- Changed the legal actions to be a list of edges remaining that have not been picked during prior timesteps.

We approached the implementation of the MST game in three varying levels of difficulty, namely: easy, medium and hard. In the easy version of the game, when deciding when to choose the next edge from legal actions, all edges that, if added, will cause a cycle have been removed from the legal actions set. Also a feature in the easy version is n-1 edge detection, where the legal actions is an empty set after n-1 edges have been selected, and thus the "game" has been terminated and the user is left with a minimum spanning tree at best, and a spanning tree at worst.

The medium implementation is the same as the easy implementation, except in termination criteria. The medium game terminates after $n-1$ timesteps, without cycle detection. Instead of simply not allowing cycle-causing edges to be chosen, we assigned hugely negative rewards to edges that cause a cycle and end the game if they are chosen. Therefore the game terminates after n-1 edges have been selected, or after an edge is selected such that a cycle has been created. We approached this implementation with the hope that our model will learn to not choose edges that cause cycles via the negative reward system, instead of hard-coding it in.

The hard implementation of the game abandons both cycle detection and n-1 edge detection. Instead, every time an edge is selected and added to the state, we check if the state of edges is a spanning tree. If not, then either add an edge or remove an edge from the existing state until a spanning tree is detected.

Figure 3. Tabular representation of game versions

Game Version	Cycle Detection	N-1 Edges Detection
Easy	Yes	Yes
Medium	No	Yes
Hard	No	No

4.3.1. Monte Carlo Tree Search

We built our game into OpenSpiel's framework so that we could use their implementation of Monte Carlo Tree Search, as well other reinforcement learning algorithms. As touched upon earlier, there are $n-1$ states in the state-space of the MST game, where the first state is the empty set and the last state is $n-1$ states after the first, and is comprised of the $n-1$ edges that make the spanning tree.

Other games follow a similar logic. Tic-Tac-Toe, Chess and Suduko for example have this common property that leads to the exponential increase in the number of actions (and

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

Figure 4. State tree of a Tic-Tac-Toe game. (Joshua Eckroth, Adversarial Search)

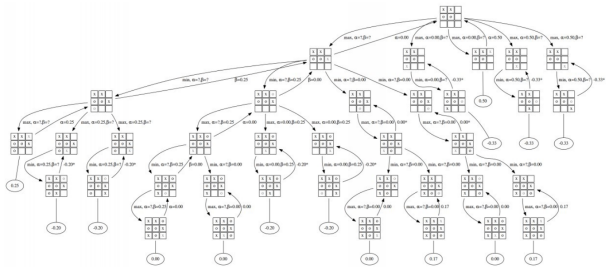
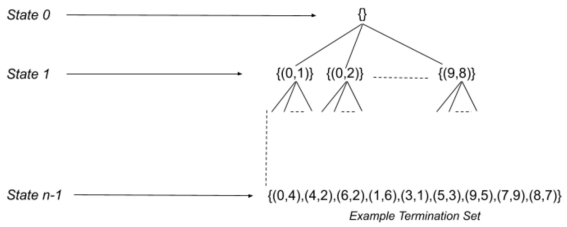


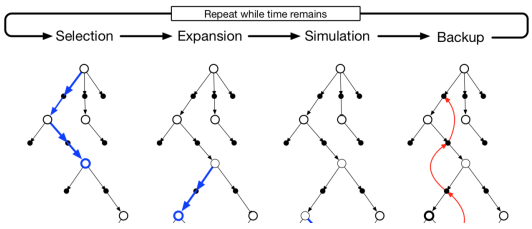
Figure 5. Example state tree of a MST game



hence states) that can be achieved in any one game. This of course means that the computing time and power required to perform such computations is also $O(n^2)$.

Monte Carlo Tree Search is used to predict the path (sequence of actions) that should be taken by the policy to reach the final winning solution, which in our case is the graph's minimum spanning tree.

Figure 6. MCTS steps (Sutton & Barto, 2018)



Expanding through all possible nodes and branches in a state tree in order to find the optimal solution will be extremely slow and, in the cases of graphs with a large node count, computationally infeasible. Therefore implementing and using a faster more efficient tree search is favorable in this case.

MCTS is an algorithm that figures out the best move out of a set of moves by selecting, expanding simulating and ultimately updating the nodes in tree to find the final solution. This method is repeated until it reaches the solution and learns the policy of the game.

Selecting

Each inter-state edge in the search tree has a weight between [0,1] that represents a probability. In this step, the node (state) with the highest probability is selected, keeping in mind that each node represents a unique collection of edges. Since the selected node has the highest probability of producing a MST at the end, we have already ruled out a vast amount of states that would not produce the MST.

Expanding

After the node is selected, we explore the children node of the selected node and again choose the most probable. Nodes that are not selected and thus not expanded are known dubbed "leaves".

Simulation

This is the essential part of the MCST algorithm - it computes the probabilities of each inter-state edge. Reinforcement learning is applied and rewards the selection of each node that produces a spanning tree close to the minimum spanning tree more than nodes that don't.

Backup

This is essentially backpropegation, ensuring that the parent nodes are connected with updated probabilities.

5. Experiments

5.1. Evaluation Procedure

PREDICTED MST LENGTH

The average predicted MST length is the sum of the $n-1$ edges that we predicted.

% ABOVE TRUE MST LENGTH

This metric was calculated by subtracting the True MST length by the Predicted MST Length and then dividing that value by the True MST Length.

6. Results

We ran experiments on graphs of size 5, 10, 20, 30, 40 and 50 nodes to see how well we could predict the actual length of the MST. The histograms show how well the GCNN+MCTS was able to preform on the "Easy Game" for 1000 random graphs with 10, 20, 30, 40 and 50 nodes respectfully. The "Percentage Difference" column in Figure 9 shows the Percent difference between what the the GCNN+MCTS predicted and the true MST values for the 1000 randomly generated graphs used for each of the histograms. The GCNN+MCTS was less then a tenth of a percent away from the True MST value on average for the 1000 graphs with 5 nodes, and only 1 %, 3.47 %, 6.77% and

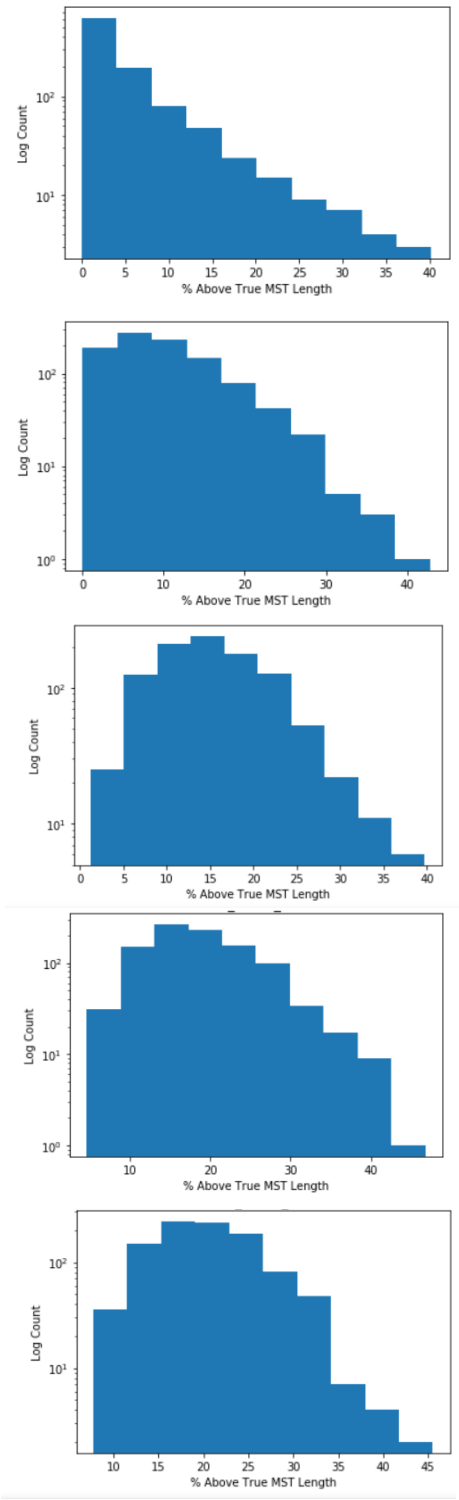


Figure 7. Histograms Of GCNN+MCTS performance in the "Easy Game" on a set of 1000 randomly generated graphs of 10 (Top) , 20, 30, 40, 50 Nodes (Bottom)

20.56% above on average for True MST Value for graphs with 5, 10, 20, 30, and 50 nodes respectively.

The GCNN+MCTS was not always able to find a spanning tree for the "Medium Version" of the game (in the "Medium Version" of the game it is not forced to output a Spanning Tree, like it is in the "Easy Version"). The Table in figure 8 shows what percent of the subset of graphs returned by the GCNN+MCTS were spanning trees for the 1000 randomly generated graphs with 5, 10, 20, 30, 40, and 50 nodes respectively.

Figure 8. Percentage that returned spanning trees (medium version)

Number of Nodes	How Many % Were Spanning Trees
5	99.8
10	78.8
20	18.1
30	4.5
40	2.1
50	1.0

Figure 9. Experiment summary statistics (easy version)

Number of Nodes	Number of Games Played	Overall Predicted Length	Overall MST	Percentage Difference
5	1000	1342.877	1342.085	+5.9e-2
10	1000	2102.204	2081.345	+1.00
20	1000	3114.540	3010.162	+3.47
30	1000	3981.768	3729.152	+6.77
50	1000	5749.253	4769.158	+20.55

7. Conclusions

We set out to see the ability of deep learning algorithms to generalize on combinatorial optimization problems. We chose to focus on the simplest problem that we could think of, the minimum spanning tree, which can be solved classically in expected linear time. We explored a few different approaches to solve the MST problem. We adapted a convolution neural network to learn the implicit structure of the MST, and used it to give predictions on the most likely edges. Additionally, we implemented an easy, medium, and hard version of Minimum Spanning Tree as a game in OpenSpiel, which we used for the deep Reinforcement Learning algorithms (DQN, Q-policy gradient) and for Monte Carlo Tree Search.

8. Future Work

Future work would include expanding our GCNN + MCTS to larger graphs of 100-1000+ nodes and seeing the performance. Additionally, we would further explore deep reinforcement learning algorithms to see their performance on this task. We would ideally like to implement graph neu-

ral network based RL-algorithms directly into OpenSpiel in order to contribute to open-source and help others trying to solve similar problems.

Acknowledgements

We'd like to thank Professors Madeline Udell, Brenda Dietrich, David Williamson, and Iddo Drori for allowing us to pursue this project, providing their council, and for pointing us in the right direction with bi-monthly meetings. They provided extremely valuable insight and are all experts in their respective fields.

References

Bresson, X. and Laurent, T. Residual gated graph convnets. *CoRR*, abs/1711.07553, 2017. URL http://arxiv.org/abs/1711.07553.

Furcy, D. and Koenig, S. Limited discrepancy beam search. In *Proceedings of the 19th International Joint Conference on Artificial Intelligence*, IJCAI'05, pp. 125–131, San Francisco, CA, USA, 2005. Morgan Kaufmann Publishers Inc. URL http://dl.acm.org/citation.cfm?id=1642293.1642313.

Joshi, C. K., Laurent, T., and Bresson, X. An efficient graph convolutional network technique for the travelling salesman problem. *CoRR*, abs/1906.01227, 2019. URL http://arxiv.org/abs/1906.01227.

Lanctot, M., Lockhart, E., Lespiau, J.-B., Zambaldi, V., Upadhyay, S., Pérolat, J., Srinivasan, S., Timbers, F., Tuyls, K., Omidshafiei, S., Hennes, D., Morrill, D., Muller, P., Ewalds, T., Faulkner, R., Kramár, J., Vylder, B. D., Saeta, B., Bradbury, J., Ding, D., Borgeaud, S., Lai, M., Schrittwieser, J., Anthony, T., Hughes, E., Danihelka, I., and Ryan-Davis, J. OpenSpiel: A framework for reinforcement learning in games. *CoRR*, abs/1908.09453, 2019. URL http://arxiv.org/abs/1908.09453.

NetworkX developer team. Networkx, 2014. URL https://networkx.github.io/.

Sutton, R. S. and Barto, A. G. *Reinforcement Learning: An Introduction*. The MIT Press, second edition, 2018. URL http://incompleteideas.net/book/the-book-2nd.html.

You, J., Ying, R., Ren, X., Hamilton, W. L., and Leskovec, J. Graphrnn: A deep generative model for graphs. *CoRR*, abs/1802.08773, 2018. URL http://arxiv.org/abs/1802.08773.

Zhou, J., Cui, G., Zhang, Z., Yang, C., Liu, Z., and Sun, M. Graph neural networks: A review of methods and applications. *CoRR*, abs/1812.08434, 2018. URL http://arxiv.org/abs/1812.08434.