Approximating Graph Algorithms with Graph Neural Networks

Maxwell Jones
Carnegie Mellon University
Pittsburgh, PA 15213
mjones2@andrew.cmu.edu

David Luo
Carnegie Mellon University
Pittsburgh, PA 15213
djluo@andrew.cmu.edu

Jocelyn Tseng
Carnegie Mellon University
Pittsburgh, PA 15213
jocelynt@andrew.cmu.edu

1 Introduction

Graph neural networks have been shown to be a powerful tool for aiding the solution of various combinatorial problems (3), including the traveling salesman problem (5), SAT (8), and maximum independent set (1). Specifically, we are using graph-based diffusion solvers to find approximate solutions graph based problems like minimum spanning tree (MST), Min cut, and single shortest path problem (SSPP). Utilizing graph neural networks to tackle graph algorithms is a task that has not garnered much attention, as the problems are not NP-hard and the small versions can be fully solved (15). In this work, we aim to aid these algorithms by using a GNN to give good heuristics/initializations for these problems. After training, we can use greedy algorithms with these heuristics to yeild solutions quickly.

For all of our problems, we will consider a problem P as a function $P : G \to \mathcal{P}(E_G)$ that takes in a graph and outputs a subset of the graphs edges. We parameterize the "cost" of this function by

$$\sum_{e \in E} w_e + \infty * \mathbb{I}[\text{invalid}(E)]$$

where nivalid(E) returns 1 if our subset is invalid. Optionally,the graph may be conditioned on some set of vertices, yeilding problem $P : G \times \mathcal{P}(V_G) \to \mathcal{P}(E_G)$. We will next discuss each problem specifically that we examine.

Minimum Spanning Tree. In the minimum spanning tree problem, a graph G is our input, and the goal is to output a set of edges $E \subseteq E_G$ minimizing the cost of the sum of edges. Here the function invalid(E) is satisfied if the edges do not satisfy the tree property (i.e. $|E| = |V_G| - 1$, and subgraph induced by E is both connected and acyclic)

Min Cut. In the min cut problem, a graph G is our input, and the goal is to output a set of edges $E \subseteq E_G$ minimizing the total cost of edges. Here the function invalid(E) is satisfied if the subgraph $G' = (V_G, E_G \setminus E)$ does not have at least 2 connected components. In other words, removing edges E does not disconnect the graph

Single Shortest Path Problem. For the single shortest path problem, we plan to approximate Dijkstra's and take in a graph G as well as a source s and target t as input (our extra conditioning), hoping to find a set $E'_{s,t}$ minimizing the total cost of edges. Here our function invalid(E) is set to true if the edges do not form a single path from s to t

2 Background

In our midway report, we consider a more naive graph based algorithm to find the shortest path between 2 points. Specifically, we consider a GNN where each nodes prediction value is a combination of its neighbors values. Specifically, let $F \in \mathbb{R}^{n \times f}$ be the current features of a graph with feature dimension f and n nodes, $A \in \mathbb{R}^{n \times n}$ be the adjacency matrix, and $W \in \mathbb{R}^{f \times f'}$ be the weight matrix with some input and output dimensions f and f'. We perform:

$$F' = \text{ReLU}(AFW)$$

twice to get our final prediction. In other words, we are applying a fully connected linear layer to our features, weighted by the graph weights. Our loss function pushes the target node t to have final output value 1.

Since each node is a strict combination of its neighbors, forcing the target node to have a final output value of 1 will in turn force its neighbors to have some output value greater than 0. Since we weight by edge weight (when multiplying with the adjacency matrix), this causes the nodes with smallest edge weight to have higher values.

For this baseline, we consider graphs with 10-25 nodes, with each set of 2 nodes having an edge with probability 50%. If two nodes have an edge, then the weight of that edge is random from 1 to 10.

After training, we start at node s and simply follow the path with highest node weight to get to t. First of all, notice that we are not directly optimizing for a shortest path objective, which could cause this method to fail for graphs with higher node count. Second, note that this method must be optimized for each individual start and end state s and t, as it has no way of training for an arbitrary start and end state.

3 Related Works

Graph Neural Networks. Graph neural networks (GNNs) are utilized to solve a wide array of tasks such as node classification, graph classification, recommendation systems, social networks, and more. They leverage the relational information encoded in the graph topology and node features to learn powerful representations that capture the underlying patterns and relationships within the data.

In DIFUSCO, anisotropic graph neural networks are the choice of network to be used as diffusion solvers for combinatorial optimization. Anisotropic graph neural networks are designed to handle graphs with heterogeneous structures and edge types. Unlike traditional graph neural networks that treat all edges equally, AGNNs consider the directional and type-specific information present in the edges of the graph (11). By incorporating this information, AGNNs can capture the varying importance and semantics of different edges in the graph, enabling more nuanced and effective learning representations for complex tasks such as tackling graph-based learning tasks (13).

AGNNs are useful for this particular task because they can produce embeddings for nodes and edges, as opposed to other GNNs that can only produce embeddings for nodes. AGNNs are used as the graph-based denoising network that takes in the noisy data, a set of nodes, and based on the problem instance we are trying to solve, will output the clean data (11). In general, graph-based denoising networks are good for tasks where not everything about the input graph is perfectly known and we aim to solve an optimization problem over properties of the graph itself in addition to the learnable GNN parameters (7; 14), making it suitable for approximating a variety of graph algorithms.

Combinatorial Optimization. A substantial effort has been put into making instances more scalable for tasks with large data. For example, Fu et al. (5) trained a message passing algorithm that is executed on subgraphs to create possible TSP instances for parts of the network. Predictions on subgraphs are then aggregated to the full graph, where a RL policy is used to help make the final TSP prediction.

When working with djikstra's algorithm and prim's algorithm, Yan et al. (15) encode the addition and minimum steps of the conventional algorithm using a neural network, and use these as subroutines in an algorithmic version of djikstra's, as opposed to requiring a model to find the entire shortest path by working directly with a graph.

The standard Bellman Ford algorithm has been used as inspiration along with GNNs for link prediction (16), a task in which the goal is to determine if any two nodes of a graph are linked. For any two

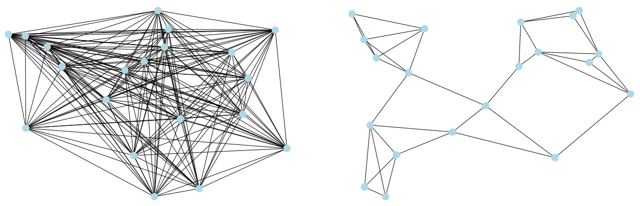

Figure 1: Examples of 20 node fully connected graph (left), and 20 node KD-Tree (right), with K = 4.

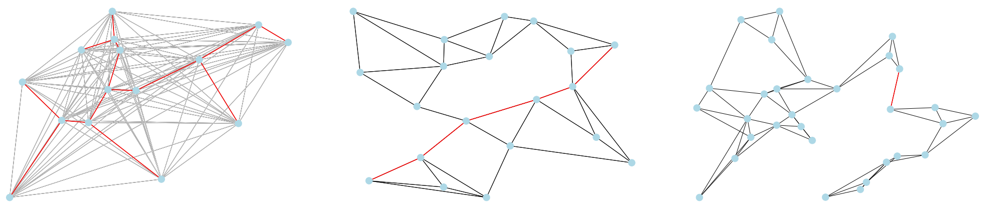

Figure 2: Examples of minimum spanning tree (left), single shortest path (middle), and minimum cut (right) problems. For SSPP, we choose the bottom left and top right nodes to be the start and end nodes respectively. Solutions to each problem are denoted by the red edges.

nodes u and v, they use a neurally network to encode each individual path P_i from u to v, then use a neural network to combine these embeddings for a final linkage representation. This task is slightly different to ours, as we hope to approximate the distance between u and v without the need for computing every path between the two.

Finally, Song (11) leverages diffusion models (10) to iteratively denoise predictions for edge weights being in or out of a TSP path. They show that this process is highly parallelizable, and produces strong results even as graph size scales.

The field of GNNs in general has made advances in scalable algorithms, using techniques like localized bidirectional propagations (4) and global aggregate feature prediction (6). Further, for tasks in which predictions on edges are required (for example predicting if edges are in a TSP, MST, or Path), classes of GNNs with edge and node embeddings have been proposed (2).

4 Methods

4.1 Dataset

Since we are tackling the graph problems, a piece of data in our dataset is a set of nodes. We create our own training and test data, where a single datum(graph) consists of N randomly chosen nodes within a 1 by 1 square. When training, we consider a training set of 16384 graphs. For validation, we generate unseen validation/test sets of 1024 additional randomized examples. For all experiments, we set the number of nodes $N = 50$.

After generating N randomly chosen node coordinates, we then add edges to create either a fully connected graph (for minimum spanning tree) or a K-Dimensional Tree (KD-Tree) (for minimum cut and single shortest path)[1]. Each edge has a weight equivalent to the euclidean distance between its two endpoints. Examples of each graph type are shown in Figure 1. Once we create the graph, we generate the labels corresponding the ground truth solution for the given problem, which are represented by a set of edges. Examples of ground truth solutions for each problem can be seen in Figure 2.

4.2 Ground Truth

4.2.1 Minimum Spanning Tree

The ground truth to solve the MST problem is Kruskal's algorithm. First, Kruskal's algorithm sorts all the edges of the graph by their weights in non-decreasing order. Then, starting with an empty spanning tree, it iteratively selects the shortest edge that does not create a cycle when added to the spanning tree. This process continues until all vertices are connected, resulting in the minimum spanning tree of the graph. Later, we introduce a modified Krukal's algorithm, where we change the initial ordering of the edges and keep all else fixed.

4.2.2 Single Shortest Path

Dijkstra's algorithm is the ground truth method for finding the single shortest path from the bottom-leftmost vertex to the top-rightmost vertex in a weighted graph with non-negative edge weights. It maintains a priority queue to greedily select the vertex with the smallest tentative distance from the source and updates the distances of its neighboring vertices if a shorter path is found. This process continues until all vertices have been visited, yielding the shortest paths from the source to all other vertices.

4.2.3 Min Cut

The ground truth algorithm to find the minimum cut of an undirected graph is the Stoer Wagner algorithm. It iteratively contracts the graph by merging vertices until only two remain. This process involves computing the minimum cut value for each contraction step and selecting the smallest cut found overall.

4.3 Model Architecture

We base our method off the DIFUSCO approach (11), by using a graph-based denoising diffusion model that learns to predict a binary $\{0, 1\}$ valued vector label representing the solution for the given problem. The diffusion process gradually adds Bernoilli noise to this vector in the forward pass, and then does a network-based denoising process in the backward pass.

More specifically, the forward process discrete diffusion is governed by current state $x_t \in [0, 1]^{N \times 2}$, which denotes the current edge distributions for N edges, and the transition matrix Q_t, where $Q_t = \begin{bmatrix} 1 - \beta_t & \beta_t \\ \beta_t & 1 - \beta_t \end{bmatrix}$, where β_t is the amount of noise introduced at timestep t. The update is applied as $q(x_t|x_{t-1}) := x_{t-1}Q_t$. In the backward process, the denoising process is done through a denoising network, which is an anisotropic graph neural network (AGNN) that operates on both node and edge features. The denoising network takes in the coordinates of the nodes, an adjacency matrix, and the candidate solution vector. The network utilizes a message passing scheme to propagate the node and edge features between layers. For the node features, the network first aggregates information from neighboring edges using SUM pooling, and then applies batch normalization followed by a ReLU activation to help stabilize training. For the edge features, the network aggregates neighboring edge features, and then applies batch normalization followed by a 2 layer multi layer perceptron (MLP).

After generating the final embeddings, the model applies a two-neural classification head to clean the output of the diffusion process. The model is trained to maximize the log-likelihood of the ground truth solution, and the loss can be expressed as:

$$L(\theta) = \mathbb{E}_{s \in S}[-\log p_\theta(x^{s^*})|s] \tag{1}$$

where x^{s^*} denotes the ground truth (optimal) solution for training instance s.

[1]We chose to use a KD-Tree as opposed to a fully connected graph for the minimum cut and single shortest path problems to avoid trivial solutions.

4.4 Training

We train using the dataset described in Section 4.1 for 20 epochs with a batch size of 32. After training, we use the epoch with the best validation loss and test on the test set. Training takes approximately 20 minutes on a single Nvidia A5000 GPU with 24 Gb of memory.

4.5 Inference and Decoding

For denoising using our learned model, we consider DDIM (9), an approach that approximates the full backward process in a shorter number of steps, with tradeoffs between matching the true backward pass and fast computation. Our diffusion models $p_\theta(\cdot|s)$ produce the final output through Bernoulli sampling, and the final score $p_\theta(x_0 = 1|s)$ is a labelingg of each edge in the graph between 0 and 1, where values closer to one indicate higher likelihood of being in the final solution. We also consider normalizing each heatmap score A_{ij} by A_{ij}/d_{ij}, where d_{ij} represents the euclidean distance between nodes i and j. We find that this helps in some instances, and provide more details in Section 6. Given both the unnormalized and normalized generated scores from our model, we then use greedy decoding strategies to create our solutions for each problem.

For MST decoding, we first rank the edges in descending order by their normalized/unnormalized heatmap scores, and then run modified kruskals algorithm on the edges in this order, by iteratively adding edges that do not create cycles.

For Single Shortest Path decoding, we run a Depth-First Search (DFS) starting from the starting (bottom left) node, each time traversing the neighboring edge with the highest normalized score. Once we reach the target (top right) node, we return the found path from the DFS.

For Min Cut decoding, similar to MST, we first rank the edges in descending order by their heatmap score. Then, we iteratively add the edges with the highest score, each time checking if the collected edges disconnect the graph using a Breadth-First Search (BFS), and stop once the edges form a cut.

4.5.1 Complexity Analysis

Let N and E be the number of nodes and edges in a graph, respectively. Note that $E \in O(N^2)$ for a fully connected graph, and, and $E \in O(KN)$ for a KD-Tree.

The MST ground truth solution utilizes Kruskal's algorithm, which is $O(N^2 \log N^2)$ for a fully connected graph. Our MST decoding process runs a version of kruskals on the heatmap scores, which is equivalently $O(N^2 \log N^2)$. However, as shown in the Results section, our decoding process encounters much less cycles compared to the ground truth solution, so our model's computational complexity enjoys a much smaller constant factor.

The SSPP ground truth solution is computed using Dijkstras algorithm, which is $O(KN \log N)$ for a KD-Tree. Our SSPP decoding process runs a DFS from the start to end node, which is $O(KN)$. Thus, we see our SSPP model has a lower computational complexity than the ground truth.

The Min Cut ground truth solution is computed using the Stoer Wagner algorithm, which is $O(KN^2 + N^2 \log N)$ for a KD-Tree. Our Min Cut decoding process takes $O(KN \log KN)$ for sorting, and $O(KNL)$ for looping through each edge and running BFS (where L is the expected number of edges found in the mincut), for an overall time complexity of $O(KNL + KN \log KN)$. Thus since L is expected to be significantly less than the total number of edges KN, our min cut model has an expected lower computational complexity than the ground truth.

5 Results

5.1 Minimum Spanning Tree

In Figure 3, we display the state of training at the beginning, middle, and end of the denoising process that our denoising GNN goes through to find a solution to the MST problem on a graph with 50 nodes. In Figure 4, we compare the number of cycles seen by our model and cost against the ground truth, Kruskal's algorithm, and also plot a histogram of the difference between the number of cycles removed by our model and by Kruskal's. The solid line represents the median value across the

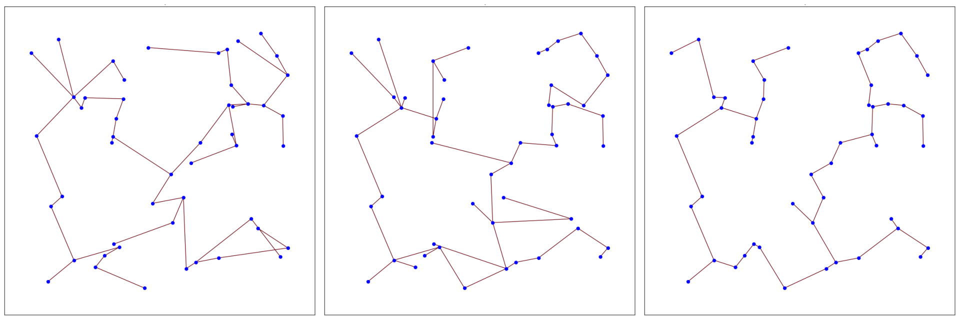

Figure 3: Decoded trees from x_t predictions at denoising steps at $t = 999$(left), 600(middle), 0(right) for 50-node MST solution. We use our model to denoise a set of randomly initialized edge weight probabilities, where the optimal solution is probability 1 for edges in the MST and 0 otherwise. Given a set of denoised edge weights, we pick edges in order from highest to lowest weight, and add edges that do not form a loop with the current set of edges. We use the DDIM(9) denoising scheduler with 50 steps during inference.

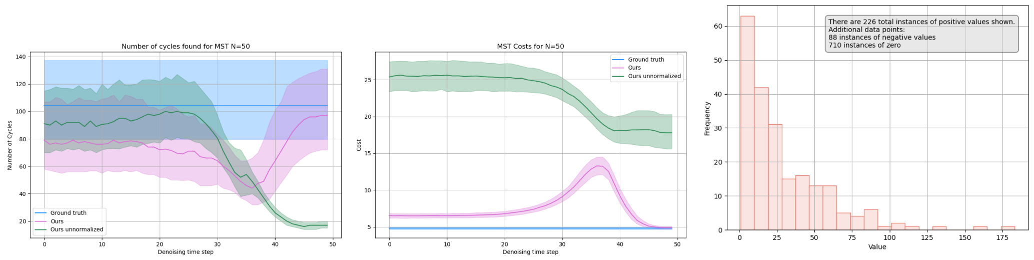

Figure 4: Plots of number of cycles and cost of our model for finding MST on graph with 50 nodes during denoising process and histogram of the difference between number of cycles removed by our model after completion of denoising steps and the ground truth (lower number of cycles is better).

samples at a particular denoising step and the ranges represent the lower and upper quartiles at that value.

5.2 Single Shortest Path

In Figure 5, we compare the number of nodes explored by our model and cost against the ground truth, Dijkstra's algorithm, and also plot a histogram of the difference between the number of nodes explored by our model and by Dijkstra's. Similar to MST, the solid lines represent median and ranges are the quartiles.

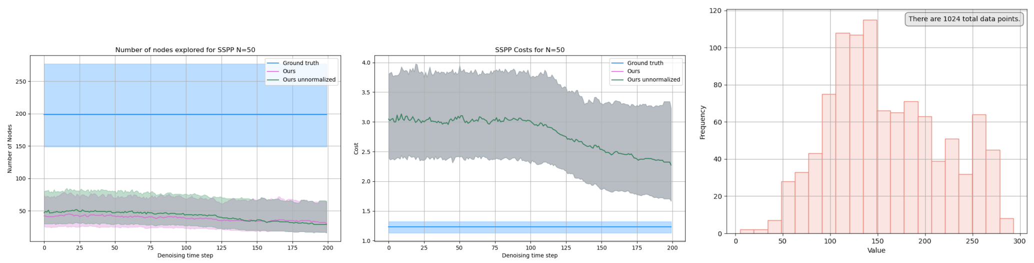

Figure 5: Plots of number of nodes explored and cost of our model for finding SSPP on graph with 50 nodes during denoising process and histogram of the difference between number of nodes explored by our model after completion of denoising steps and the ground truth (lower number of nodes is better).

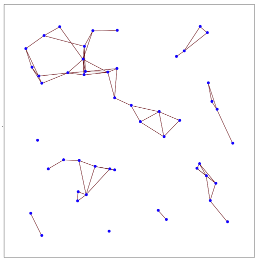
Figure 6: Approximate minimum cut found from heatmap scores using greedy decoding process. The approximate cut contains much more edges compared to the ground truth.

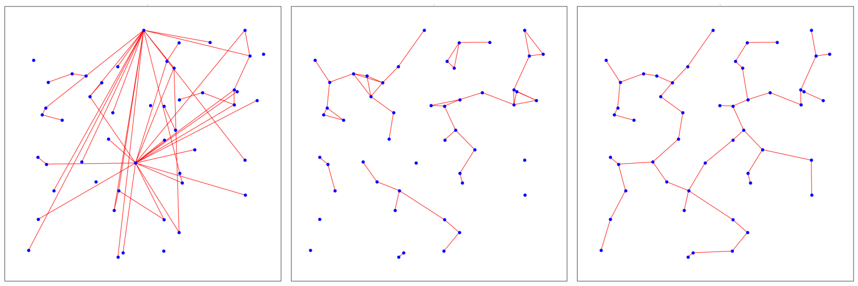
Figure 7: We ablate the different parts of our inference process for MST given some set of edge weight likelihoods. In the first case (left), we simply pick the n - 1 edges with highest likelihood to be in the MST from the model prediction. In the second case (middle), we normalize each likelihood by the edge weight itself (in this case euclidean distance) and pick the n - 1 edges with highest likelihood. In the final case (right), we consider Kruskal's algorithm with an ordering equal to likelihoods of edges normalized by their weights.

5.3 Min Cut

In Figure 6, we display an example of an approximate minimum cut solution found by our model.

6 Discussion and Analysis

6.1 Minimum Spanning Tree

In Figure 4, we plot denoising step vs number of cycles found by modified Kruskal's algorithm (left) as well as denoising step vs MST cost (right). Here we are using the edge weights predicted from the denoised x_0 prediction from each timestep, as opposed to the noisy edge weights x_t. In the optimal case where all the MST edges are ordered perfectly as the top $n - 1$ edges, the cost will be equal to the MST cost, and the number of cycles will be 0. During our decoding process, the unnormalized edge weights decline to a very low number of cycles encountered, but stay fairly high in MST prediction. We surmise that this is because certain nodes in the graph give all edges high likelihood (see Figure 7, left), causing less cycles to be found. Notice that if the top $n - 1$ edges all have a shared node, then the tree is a star graph (which has high cost), and modified Kruskal's algorithm find no cycles (since all edges are adjacent).

In the normalized case, we see a similar pattern for intermediate edge weights, followed by an increase in cycle number back up to the gt number. We also see that our cost decreases to the ground truth cost. Interestingly, when plotting ground_truth_cycles_found - our_cycles_found (Figure 4), we find almost exclusively nonnegative values. This indicates that our method either performs the same as the ground truth w.r.t. number of cycles (75% of the time) or performs more optimally (25% of the time), decreasing the number of cylces found when using modified Kruskal's.

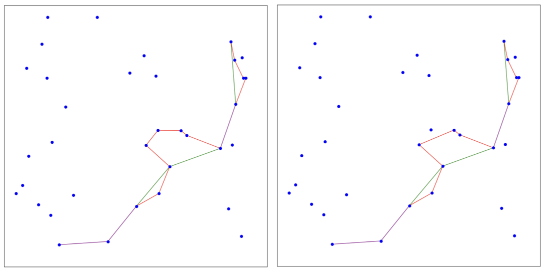

Figure 8: Examples of shortest paths found using normalized (left) and unnormalized (right) methods

6.2 Single Shortest Path

For Single Shortest path, we find that normalizing does not make a huge difference in prediction as for MST. This may be due to the greedy nature of the algorithm, where only a few edge values are being compared against each other at any decoding step. In Figure 8, we notice that the paths generated by the normalized and unnormalized graphs look very similar, with the exception of a single, slightly longer deviation taken by the normalized decoding path at step 6 of the path given by our solution. Notice that the normalized path chooses a shorter edge over its longer counterpart, which intuitively makes sense as longer (and thus heavier) edges get downweighted by the normalization term $1/w_{ij}$.

Overall, we see that our decoded solution does get shorter over time, however it still performs around 2x worse than the final solution. This could be due to the fact that a single deviation from the ground truth optimal path can cause large increase in the final decoded path, as we use a simple DFS with priority to higher weighted edges as a decoding mechanism. Instead of a DFS, we could instead use a small lookahead search to gain better performance, like Monte Carlo Tree Search (12). In fact, the original difusco paper (11) uses MCTS for better decoding of edge weights into TSP solutions, so it is a powerful tool that we leave to future work.

6.3 Min Cut

For Min Cut, we found that our decoding process struggled greatly to generate low-cost cut solutions from our generated heatmap. From an empirical analysis, we saw that if a node had K neighboring edges, the heatmap scores would tend to be high for $K-1$ edges, and low for the Kth edge. As a result, our greedy decoding process would only partially remove the neighboring edges from a node before removing edges from other nodes, causing the number of edges found in the cut to be much higher than the ground truth. An example of a cut found my our approximation deoding process is shown in 6.

7 Teammates and work division

Maxwell: Coded, trained, and fine-tuned the diffusion models. Ran multiple experiments and ablation studies for each graph problem. Generated denoising step visualizations. Wrote the Introduction, Background, Combinatorial Optimization Related Works, Dataset, Training, and part of the Discussion and Analysis sections.

Jocelyn: Generated visualizations, metrics, and evaluation for each graph problem, including computational complexity cost comparisons between our architecture and ground truth solutions. Wrote the GNN Related Works, ground truth Models, and minimum spanning tree and single shortest path Results sections.

David: Coded the data generation process and decoding process of the models. Generated example graph and solution visualizations. Wrote the Model Architecture, Inference and Decoding, Complexity Analysis, min cut results, and part of the Discussion and Analysis sections.

8 Access to your Code

Codebase link

References

[1] Ahn, S., Seo, Y., Shin, J.: Learning what to defer for maximum independent sets. In: International conference on machine learning. pp. 134–144. PMLR (2020)

[2] Bresson, X., Laurent, T.: An experimental study of neural networks for variable graphs (2018)

[3] Cappart, Q., Chételat, D., Khalil, E.B., Lodi, A., Morris, C., Veličković, P.: Combinatorial optimization and reasoning with graph neural networks. Journal of Machine Learning Research **24**(130), 1–61 (2023)

[4] Chen, M., Wei, Z., Ding, B., Li, Y., Yuan, Y., Du, X., Wen, J.R.: Scalable graph neural networks via bidirectional propagation. Advances in neural information processing systems **33**, 14556–14566 (2020)

[5] Fu, Z.H., Qiu, K.B., Zha, H.: Generalize a small pre-trained model to arbitrarily large tsp instances. In: Proceedings of the AAAI conference on artificial intelligence. vol. 35, pp. 7474–7482 (2021)

[6] Kong, K., Chen, J., Kirchenbauer, J., Ni, R., Bruss, C.B., Goldstein, T.: Goat: A global transformer on large-scale graphs. In: International Conference on Machine Learning. pp. 17375–17390. PMLR (2023)

[7] Rey, S., Segarra, S., Heckel, R., Marques, A.: Untrained graph neural networks for denoising. Institute of Electrical and Electronics Engineers (2023)

[8] Selsam, D., Bjørner, N.: Guiding high-performance sat solvers with unsat-core predictions. In: Theory and Applications of Satisfiability Testing–SAT 2019: 22nd International Conference, SAT 2019, Lisbon, Portugal, July 9–12, 2019, Proceedings 22. pp. 336–353. Springer (2019)

[9] Song, J., Meng, C., Ermon, S.: Denoising diffusion implicit models. arXiv preprint arXiv:2010.02502 (2020)

[10] Song, Y., Ermon, S.: Generative modeling by estimating gradients of the data distribution. Advances in neural information processing systems **32** (2019)

[11] Sun, Z., Yang, Y.: Difusco: Graph-based diffusion solvers for combinatorial optimization. Advances in Neural Information Processing Systems **36** (2024)

[12] Świechowski, M., Godlewski, K., Sawicki, B., Mańdziuk, J.: Monte carlo tree search: A review of recent modifications and applications. Artificial Intelligence Review **56**(3), 2497–2562 (2023)

[13] Tailor, S., Opolka, F., Lio, P., Lane, N.: Do we need anisotropic graph neural networks? International Conference on Learning Representations **10** (2022)

[14] Tenorio, V., Rey, S., Marques, A.: Robust graph neural network based on graph denoising. Asilomar Conference on Signals, Systems, and Computers (2023)

[15] Yan, Y., Swersky, K., Koutra, D., Ranganathan, P., Hashemi, M.: Neural execution engines: Learning to execute subroutines. Advances in Neural Information Processing Systems **33**, 17298–17308 (2020)

[16] Zhu, Z., Zhang, Z., Xhonneux, L.P., Tang, J.: Neural bellman-ford networks: A general graph neural network framework for link prediction. Advances in Neural Information Processing Systems **34**, 29476–29490 (2021)