Graph Neural Networks for NP-Hard Combinatorial Optimization Problems (Traveling Salesman Problem)

Everett Knag, Justin Saluja, Chaitanya Srinivasan, Prakarsh Yadav *
Carnegie Mellon University
Pittsburgh, PA 15213
`{eknag, jsaluja, csriniv1, pyadav}@andrew.cmu.edu`

Abstract

This work focuses on the application of deep learning on combinatorial optimization problems, in which finding the optimal solution proves to be computationally intractable for large inputs. We formalize these problems using graph theory and leverage the expressive power of graph neural networks to find acceptable solutions for relevant problem instances instead of finding optimal solutions for all problems instances. To do this, we use heuristics that are designed to be relevant specifically for the Traveling Salesman Problem. The design of heuristics is much like feature engineering in that it needs expert knowledge and time consuming hand tuning. Instead of feature engineering, we apply a deep learning approach to automate this process. We compare the performance of two polynomial time deep learning approaches, Joshi et al. [1] and DPDP [2], against the performance of the Concorde TSP exact solver and naive greedy search heuristic. We provide comparisons of reimplementations of the original GNN-based approaches and find improvements in performance by expanding upon the published methods.

1 Introduction

A decision problem D is in the set of NP-Hard problems if every decision problem in NP, the set of polynomial time verifiable problems, can be reduced to D. This renders the solutions at large scales infeasible to solve using traditional algorithms. NP-hard Combinatorial Optimization (CO) problems arise in various fields of scientific study, such as bioinformatics, operations research, electrical engineering, *etc.* [3]. The algorithmic solution to these problems at large scale and in their worst case struggles with computational cost and generalizability. This limitation of heuristic algorithms has attracted substantial interest from the Machine Learning domain to learn better algorithms to solve these problems.

CO problems like the Travelling Salesman Problem (TSP) can be interpreted as a graph optimization problem. The aim is then to find the optimal graph structure which solves the TSP problem. Many NP-Hard problems can be reduced to such graph optimization problems. Machine Learning-based methods can find learning algorithms which can solve any of these optimization problems. The developments in Deep Neural Networks (DNN) have inspired their application to solving the NP-Hard problems [4]. However, due to the constraints of DNN architecture, the DNN based methods cannot interpret the underlying graph-based structure of the NP-Hard problems and struggle with generalization.

This constraint can be overcome by the Graph Neural Networks (GNN) [5]. GNNs are deep learning methods which operate in the graph domain and can be used as a graph analysis or optimization method. By defining the NP-Hard problems in terms of a graph, it is possible to use GNNs to predict the likelihood of each vertex belonging to the optimal solution [6]. In this work, we propose to investigate the potential of GNNs to find the optimal solution for NP-Hard combinatorial optimization (CO) problems.

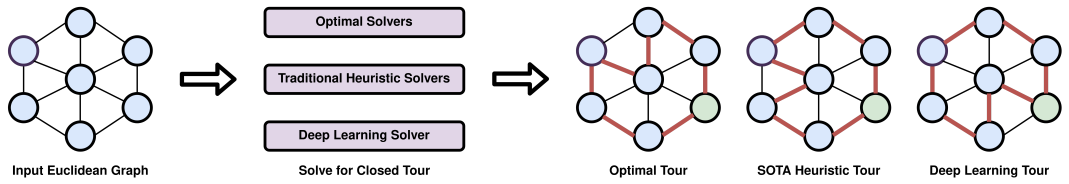

Figure 1: Graph Neural Network Pipeline and Comparison.

2 Review of Literature

TSP has been a central CO problem since the 1930s. As a particularly famous NP hard problem, significant traditional algorithms research has been conducted. Several high performance solvers exist, but the asymptotic complexity of all existing exact solvers means that the problem quickly becomes infeasible for large graphs. Thus, many heuristic solvers also exist. These heuristic solvers aim to significantly reduce the asymptotic complexity of the problem while maintaining a small optimality gap. Recent applications of GNNs to TSP have yielded promising results on relatively small graphs on the order of less than 100 vertices [7]. A thorough review of baseline Graph Neural Network (GNN) models for combinatorial optimization (CO) is two-fold. We must examine efforts involving both machine learning for CO and GNNs individually and then jointly.

Graph Neural Networks have recently regained popularity as the dominant machine learning method for graph structured input. Prominent architectures include graph convolutional networks [8], graph attention networks [9], and inductive representation learning [10]. Numerous spectral methods have also been proposed. Given the recent traction, an abundance of survey papers touch on the latest GNN techniques and application on various reasoning tasks [11, 12, 5].

Without focusing too much on graph structured input, Bengio et al. [13] gives a broad overview on machine learning methods for CO. Machine learning for mixed integer programming [14] using branching and reinforcement learning [15, 16] for CO are also considered. We observe practical engineering advances like networking problems for telecommunications that focus on heuristically driven approaches [17].

Since there are many practically relevant NP hard CO problems, this will be the main direction of this research. Broadly speaking, the class of algorithms in this domain aim to prioritize one of two goals. The main goal is to find good feasible solutions, with the secondary goal to certify optimality/prove infeasibility. In either case, GNN's provide a convincing abstraction for representing problem instances and have been shown to certify optimality more efficiently and guide algorithms toward feasible solutions faster. Looking ahead, there is room to investigate algorithmic reasoning in the context of designing and training GNNs that enable improved interpretability [18].

Both offline- and online-learning approaches have been successful in efficiently solving small instances of the TSP. Learning-based approaches for approximating CO problems have been evaluated on fixed-sized graph datasets of arbitrary structure [3]. A small subset of such approaches have been successful at scale. Considering instances of CO problems can be framed as graphs, graph neural network (GNN) approaches have become the current state-of-the-art for their scalability to larger instances and generalizability to various subclasses of CO problems. In particular, attention mechanisms in GNNs have become favorable for their flexibility in input size and ability to discriminate relevant signal in the input. Veličković et al. proposed an attention mechanism as a one-layer feedforward NN parametrized by weights that are applied to every node in the input graph[9]. Kool et al. applied this approach to CO in the context of the following problems: TSP, vehicle routing, orienteering, prize collecting TSP (PCTSP), and stochastic PCTSP[7]. The authors found that attention models achieve higher learning and computational efficiency than other DL approaches. They contend that their models could still be improved with the addition on heuristics and backtracking functionalities, which remains an open area of investigation. Graph attention networks, to our best knowledge, have not been applied to other classes of CO problems. Vanilla GNNs have been applied in strong first steps to approximate solutions to boolean satisfiability problems, but could theoretically be improved by incorporating an attention mechanism[19]. In this paper, in addition to reimplementing the Joshi et al and DPDP GNN-based solvers, we expand on these approaches by analyzing their performances over varied hyperparameter configuration, which were largely unexplored in the publications.

3 Contribution

Definitions

1. **Graph** - A graph $G = (V, E)$ is a tuple of the set of Vertices $V = \{v : v$ is an vertex in G$\}$ and the set of Edges $E = \{e_{ij} = (v_i, v_j) : i, j \in [|V|], e_{ij}$ is an edge in G$\}$.
2. **Undirected Complete Graph** - An undirected complete graph is a graph G in which $E = \{e_{ij} = e_{ji} = \{v_i, v_j\} : i, j \in [|V|]\}\}$
3. **Weighted Graph** - A weighted graph $G = (V, E, w)$ is a graph such that $w : E \to \mathbb{R}$
4. **Euclidean Graph** - A Euclidean Graph $G = (V, E, w)$ is an undirected weighted complete graph in which $V = \{v : v \in R^d, v$ is an vertex in G$\}$, and $w(e_{ij}) := d_2(v_i, v_j)$, where d_2 represents the Euclidean metric.
5. **Walk** - A walk is a sequence of vertices $v_{k_1}, v_{k_2}, ..., v_{k_n}$, $k_i \in [|V|]$.
6. **Tour** - A tour is a walk in which all edges are unique.
7. **Closed Tour** - A closed tour is a tour that begins and ends with the same vertex. For example, v_1, v_3, v_2, v_1.

Definition - TSP

In this paper, we focus on Euclidean tour-TSP. Euclidean tour-TSP or TSP for short is a function defined below:

1. let $\mathcal{G}$ denote the set of Euclidean Graphs.
2. let $T_c(G)$ be the set of all closed tours on a given graph G.
3. $TSP : \mathcal{G} \to T_c(\mathcal{G})$
4. $TSP(G) := argmin\{w(T) : T \in T_c(G)\}$
   where $w(t)$ is the sum of the edge weights on closed tour T.

That is, $TSP(G)$ is a list of vertices (coordinate) such that $TSP(G)$ is a closed tour where total travel distance is minimized.

Concorde TSP Solver

The Concorde TSP Solver has been shown to exactly solve instances of the TSP for graphs ranging in size up to 85,900 nodes. Concorde employs a variety of hand-engineered heuristics to find the tour of any given graph with minimal total Euclidean distance. In its worst case behavior, Concorde has an asymptotic time complexity of $O(|V|^2 2^{|V|})$ to find the exact solution, which is inefficient especially for large graphs. We utilized Concorde to benchmark the optimal total distance of tours and also provide labels for the optimal paths of unannotated graphs. These labels could then be applied to measure the loss of GNN approaches for TSP on unseen data. We additionally recorded the mean per-iteration runtime of Concorde on the unseen data to compare its efficiency with those of a naive, greedy approach and GNN methods.

Greedy Search Heuristic

The Greedy Search approach is a local search heuristic in which the tour is initialized with the 0th index node and iteratively updated by choosing the nearest unique neighboring node without crossing a distinct edge more than once. The time complexity of this approach is in $O(|V| + |E|)$. Since a Euclidean TSP graph instance is fully connected, the time complexity is $O(|V|^2)$. This method is naive and outputs suboptimal tours for large graphs by expectation. We measured the Greedy Search Heuristic performance on the test set data to identify whether deep learning-based approaches offer improvements in optimal tour total distance.

Model Formulation

Both the Graph Convolutional Network and Graph Attention Network are two prominent state of the art Graphical methods for solving TSP problems. We chose to sample these and rigorously compare the performance to from appropriate baselines. After initial testing, we concluded that the rich information encoded in the heatmap in Joshi's work [1] was a viable point to expand on. The recently published paper of Deep Policy Dynamic Programming [2] does this expansion and achieves near perfect accuracy. We thought that both a re-implementation and an expansion of this paper would be the perfect follow on work. The details of the tests are outlined in the following sections. Some examples of explicit measures of comparing these methods include comparative beam width and runtime tests that were lacking from the original literature. Although we abandon the Graph Attention Model, due to it's relatively poor performance compared to the other methods outlined, details about this method can be found in the Appendix.

Model Architecture - Graph Convolution Network

As described by Joshi *et al.*, the length of a tour for a permutation π is defined as,

$$L(\pi|s) = ||x_{\pi(n)} - x_{\pi(1)}||_2 + \sum_{i=1}^{n-1} ||x_{\pi(i)} - x_{\pi(i+1)}||_2$$

To prepare the input to the model, the two-dimensional coordinates are embedded into h-dimensional features by following the relationship,

$$\alpha_i = A_i x_i + b_i$$

Here, $A_1 \in \mathbb{R}^{h\times 2}$. The edge input features are defined as,

$$\beta_i = A_2 d_{ij} + b_2 || A_3 \delta_{ij}^{k-NN}$$

Here, $A_2 \in \mathbb{R}^{\frac{h}{2}\times 1}$, $A_3 \in \mathbb{R}^{\frac{h}{2}\times 3}$ and $\cdot||\cdot$ is the concatenation output. The edge distance, d_{ij}, is $\frac{h}{2}$ dimensional and the indicator function of edge, δ_{ij}^{k-NN}, is one when the nodes i and j are neighbors.

Let x_i^l denote the node feature vector and e_{ij}^l denote the edge feature vector at layer l with node i and edge ij. The node and edge feature vector at next layer are compute as

$$x_i^{l+1} = x_i^l + ReLU(BN(W_1^l x_i^l + \sum_{j\sim i} \eta_{ij}^l \odot W_2^l x_j^l))$$

here η_{ij}^l is defined as $\frac{\sigma(e_{ij}^l)}{\sum_{j`\sim i`} \sigma e_{ij`}^l} + \epsilon`$.

$$e_{ij}^{l+1} = e_{ij}^l + ReLU(BN(W_3^l e_{ij}^l + W_4^l x_i^l + W_5^l x_j^l))$$

in these equations $W_i \in \mathbb{R}^{h\times h}$, σ denotes the sigmoid function, ϵ is a small value, $ReLU$ is the rectified linear unit and BN is batch normalization. The model is initialized as $x_i^0 = \alpha_i$ and $e_{ij}^0 = \beta_{ij}$.

The last layer outputs edge embedding e_{ij}^L which is used to calculate the probability of assignment of an edge to the optimal solution, p_{ij}^{TSP}. This probability is computed by using a Multi Layered Perceptron, which is defined as,

$$p_{ij}^{TSP} = MLP(e_{ij}^L)$$

The predicted tour π is converted to an adjacency matrix, where each element denotes the presence or absence of an edge as part of the optimal solution. To compute the loss for this defined model, for the ground truth TSP tour permutation π, an adjacency matrix is defined. The edges of this adjacency matrix denote the presence or absence of the given edge in the optimal tour of TSP. The weighted binary cross entropy is minimized between this ground truth adjacency matrix and the predicted heat map from the model.

Model Architecture - Deep Policy Dynamic Programming

Deep Policy Dynamic Programming (DPDP) uses the graph neural network derived heatmap from Joshi et al. [1] (see above section) to predict a optimal solution based on the heatmap of promising edges. The original graph is pruned to only include high heat edges above a threshold, which is a tunable hyperparameter. The edges that were kept are then used to derive a policy for scoring partial solutions in the DP algorithm. An initial empty solution is initialized and the solution expands the beam on every iteration. If a dominated path is reached, then it removes it from the solution. Hence, a solution will always be uniquely defined by the ordered sequence of nodes. Then the Dynamic Programming algorithm constructs the final solution by backtracking the actions used to expand the solutions. To note the time complexity, a brute force solver is factorial $O(n!)$, dynamic programming is exponential $O(n^2 \cdot 2^n)$, and DPDP is linear depending on beam size $O(Bn)$.

We start with an initial empty solution. In the next iteration beam search expands all solutions on the beam. Next, it removes all dominated solutions. It is important to note here that the dynamic programming has stored partial tours it has seen before. Consider two partial tours that start and end and the same respective nodes. If one partial tour is of minimum total length found, we say that path dominates over other partial tours. Therefore, the partial tours that have been dominated will never be part of the optimal solution so no further branching needs to be done on these sub optimal tours. The beam search will continue to expand on the most optimal tour it has found thus far, pruning the dominated solutions. This is to encourage finding the true optimal solution. Then it selects the top B best solutions according to some scoring policy (below) to define the beam for the next iteration.

More formally the DP algorithm is defined over a graph with n nodes and the distance c_{ij} is calculated between each node. To develop the optimal solution a sequence of actions a is taken. This introduces a series of variables that are to be tracked, namely, $cost(a)$, total $cost$ (distance), $current(a)$, current node, $visited(a)$ and set of visited nodes. The initial solution at, time $t = 0$, is defined as empty solution with $cost = 0$, $current(a) = 0$ and $visited(a) = \{0\}$. These state variables are updated incrementally by using the equations:

$$cost(a') = cost(a) + c_{current(a),a_t}$$
$$current(a') = a_t$$
$$visited(a') = visited(a) \cup \{a_t\}$$

The solution dominance is defined as having a solution with, $visited(a') = visited(a)$, $current(a') = current(a)$ and $cost(a') \leq cost(a)$. This defines one $DPstate$ as a tuple, $(visited(a), current(a))$.

The heatmap, which indicates the presence of an edge to be involved in the optimal solution, is calculated from the Joshi et al. GCN. The beam search uses a slightly different scoring policy to select beams than just the heat of edges. The heatmap, $\hat{h}_{ij} \in (0, 1)$ for each edge (i, j) in graph, is used in addition to what is known as a heat *potential* in a linear combination with the heat.

$$score(a) = heat(a) + potential(a)$$

Here, $heat(a)$ and $potential(a)$ are defined as,

$$heat(a) = \sum_{i=1}^{t-1} h_{a_{i-1},a_i}$$
$$potential(a) = potential_0(a) + \sum_{i \notin visited(a)} potential_i(a)$$

where, $potential_i(a) = w_i \sum_{j \notin visited(a)} \frac{h_{ij}}{\sum_{j'=0}^{n-1} h_{j'i}}$ and w_i is potential weight defined by $w_i = \max_j h_{ij} \cdot (1 - 0.1(\frac{c_{i0}}{\max_j c_{j0}} - 0.5)$. The reason for this potential term is that some nearby nodes may be skipped if other edges have high heatmap values. These skipped nodes have to be visited later meaning that it assumes a similar pitfall as the greedy search algorithm where it can reach a 'dead-end' at the end of its tour. To avoid the 'greedy pitfall' the potential term is added.

At the time of writing this report, the paper code has become recently available from the authors. We were able to verify the results from the paper and cross checked with the implementation of the released code with appropriate cost metrics and optimality gaps.

4 Experimental Evaluation

Dataset

The training dataset consists of one million fully connected graphs with optimal tours identified by Concorde. The testing dataset consists of ten thousand fully connected graphs with no optimal tours provided. We generated these optimal tours by running Concorde to provide labels for the GNNs. Each of the graphs has 100 nodes that were generated uniformly randomly. The nodes lie in space $[0, 1] \times [0, 1]$.

Experiments

After a thorough review of the literature and the available libraries for modeling GNN architectures, we concluded that there are two promising approaches which use GNNs for CO. The first approach proposed by Kool *et al.*[7], introduces an Graph Attention Network model which is trained by reinforcement. The second approach is proposed by Joshi *et al.* [3], which introduces a Graph Convolutional Neural Network (GCNN) approach to solve the CO problems, specifically TSP. These two approaches have been highly influential in extending the application of GNNs to solve CO problems. Both of the works present the CO problems, such as TSP, as a graph problem and make use of the GNNs to extract significant features from the edges and nodes. However, these approaches differ greatly in their motivation for the selection of the desired GNN architecture, as one uses attention with reinforcement and the other uses graph convolution.

Description and validation of baseline

As discussed in the literature, the gap between the optimal solution of a TSP by the heuristics based solvers, like Concorde, and the GNN predicted solution of the same TSP is considered as a metric to evaluate the performance of the various GNN based approaches to solve CO problems. Both, Attention model proposed by Kool *et al.* and GCN model proposed Joshi *et al.* presented results which claimed that their method was markedly better than existing approaches at finding the solution to TSP with minimal gap from the optimal solution. However, when compared against each other, both the approaches had very similar performance. For instance, in case of a TSP with 20 nodes, the gap from optimal solution was at **0.08%** for attention network and **0.10%** for the GCN. In case of a TSP with 100 nodes, the gap from optimal solution was at **2.26%** for attention network and **2.11%** for the GCN. Based on the results for these methods we could not definitely claim which method was superior. Therefore in our work, we have considered both the approaches as baselines to compare the performance of our proposed model. In addition, to obtain the optimal solution for a TSP we will also use heuristics based solvers, like Concorde, to obtain the ground truth for the optimal TSP path. These ground truth will serve as the reference against which we will evaluate the GNN baselines and our proposed model.

Evaluation metrics

We identified the optimal tours for graphs in the test set using Concorde, an exact TSP solver. The tours outputted by Concorde were treated as true labels for each of the graphs to measure the divergence between tours outputted by the model and the optimal tour. We computed the mean euclidean distance across all tours to directly compare the performance of the Joshi et al., DPDP, and Greedy Search approaches (Table 1).

Baseline tests for the Graph Attention Network were run on a NVIDIA T4 GPU and a custom Intel Cascade Lake CPU. Exhaustive tests with multiple seeds were performed on graph sizes of 20 nodes using the same hyperparameters specified in [7]. We found an improvement in time per epoch from 5:30 min/iteration to 4:32 from upgrading GPUs from the 1080Ti used in the paper. Training time lasted for 100 epochs with data generated on the fly. Using the Attention Model (AM), we were able to replicate the same results presented in the paper. The objective cost bounced around 3.83-3.87 by epoch 40 but eventually converged to 3.85 with the same optimality gap for the greedy decode.

The tests for Graph Convolution Network (GCN) were run on a NVIDIA 1080Ti GPU and 10xIntel Xeon processors. The computational resources were similar to the resources used by Joshi et al. For

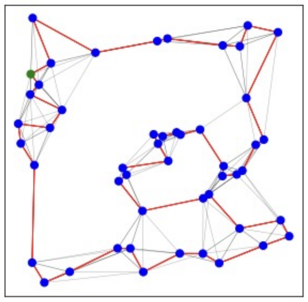

Figure 2: For a Travelling Salesman Problem with 50 nodes the optimal solution is shown by the red trace and the starting node is shown in green.

TSP size of 20 nodes and 50 nodes, we trained the GCN for 50 epochs to obtain the optimal gap close to the reported values. The optimal solution for a TSP of 50 nodes is shown in Figure 2. To ensure that we are able to reproduce the results in [3], Table 1, we resorted to using the hyperparameters specified in the paper, such as the network architecture of 30 Graph Convolution layers and 3 MLP layers with hidden dimensions of 300, learning rate of 0.0001. For the learning rate scheduler we followed the scheme specified in the paper where, if the validation loss is not reduced by a least 1% the learning rate is reduced by dividing with 1.01. By using these hyperparameters and barring minor time variations per epoch, we were able to reproduce the results. The optimal tour length for a TSP with 20 nodes was reported to be 3.86 with a gap of 0.06% from optimal solution. From the baseline GCN we had trained we obtained a best tour of length 3.948 with a gap of 0.09% from the ground truth of optimal tour length of 3.943. In case of TSP with 50 nodes the optimal tour length was 5.87 with a gap of 3.10%. From our best trained model we obtained a tour of length 6.161 with a gap of 3.38% from the optimal tour length of 5.95. In addition, we evaluated the gain in performance of the GCN model, in terms of tour length and model loss, as a function of the number of epochs trained (Figure 3).

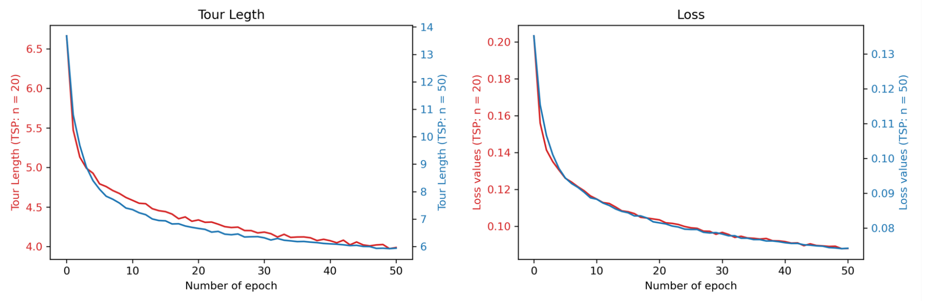

Figure 3: The plot of the optimal tour length as predicted by the model. Red plot shows tour length for TSP of size 20 and blue plot shows tour length of TSP of size 50. The plot of the model loss as a function of number of epochs . Red plot shows the loss for TSP of size 20 and blue plot the loss for TSP of size 50.

The code for DPDP was recently released on Github and was subsequently mixed in with our DPDP code. The main training loop, visualizations and various utility functions were coded from scratch in support of the already open source code. The code was also modified to support the additional tests outlined in the results section. Functions that had been optimized beam search, dynamic programming, and other utility functions were mixed into our repository from the open source code. Further modifications have also been made from the scratch code that were inspired by the open source code so that in the future in can start to support multi-threading and other highly optimized libraries. The code to install the Concorde solver was referenced from the Kool et al. Github page[7]. The implementation of GCN model as presented by Joshi et al. [1], was inspired by their GitHub repository. The main training loop was coded from scratch and integrated with the previously mentioned pipeline. Additional modifications to the network were made to ensure that it was compatible with the custom TSP generator and dataloader we had written. To compare the performance of GCN model with beam search against the DPDP method, necessary modifications were made to extract the heatmaps from the model and the evaluation functions, such as beam search and tour length calculation were restructured. The GCN model training pipeline is presented as a Jupyter notebook and the evaluation method for the heatmaps against DPDP is presented as an executable script. We wrote original code to format data into an acceptable format for Concorde, run the Concorde solver across all graph instances, and benchmark the mean optimal tour length and time-per-iteration. The greedy search heuristic code was written from scratch, along with the necessary helper functions to run the algorithm on graph instances and retrieve benchmark statistics.

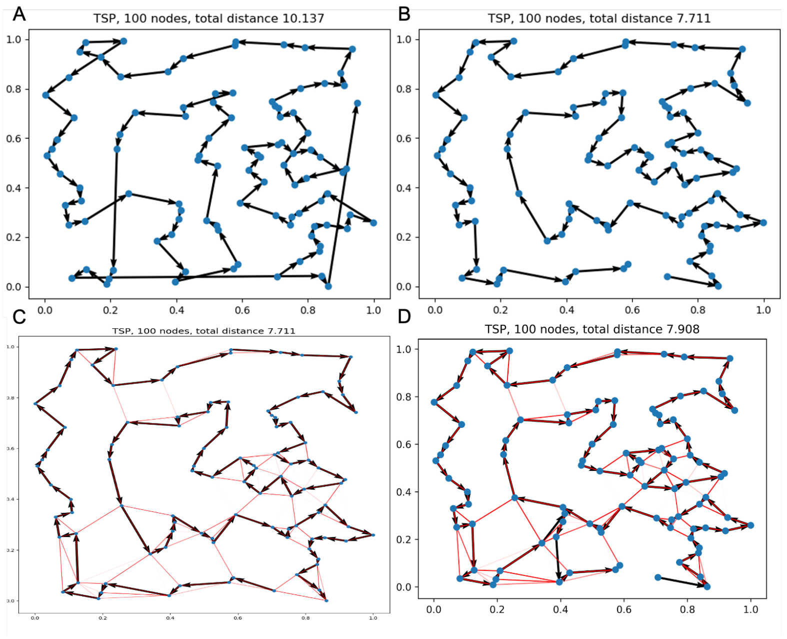

Figure 4: A TSP of 100 nodes which is solved by 4 separate methods. A. The tour found by Greedy search of nearest neighbor, with a total distance of 10.137. B. The optimal tour found by the Concorde solver with total distance of 7.711. C. The tour predicted by the DPDP method with distance of 7.711. D. The tour predicted by the Graph Convolution and Beamsearch method as described by Joshi et al.

5 Results

Our work thus far has introduced two prominent model architectures in Graph Neural Networks, the Graph Convolution Network and Graph attention. Recent literature introduces the [2] Deep Policy Dynamic Programming (DPDP) as a framework for solving vehicle routing problem. This new framework utilizes both of our proposed baselines in novel ways. More specifically, DPDP uses the graph neural network proposed in Joshi et al. [1] to preprocess the problem instance into a sparse heatmap of promising edges from which a policy is derived. We show that this is the most promising avenue for achieving SOTA results, using GNNs, that are competitive to optimal solvers.

To assess the performance of DPDP at solving TSP we compare it against the Graph Convolution approach described by Joshi et al. The framework for DPDP was made publicly available only recently, however, we have managed to rework it to be compatible with our custom variant of the Graph Convolution model and here we report the results we obtained by using both DPDP and standard Graph Convolution with Beamsearch. We verified the results obtained from the paper by Kool et al. that introduces DPDP, [2]. We were able to reproduce the results in this work and we report our observations in Table 1. We used four method for solving TSP of 100 nodes. This evaluation set had 10,000 sample TSP which were first solved using Concorde solver. The output of Concorde was considered as the optimal solution of a given TSP. We compared all other predicted tours for the TSP against the output of Concorde, Figure 4. A sample tour of TSP as predicted by Concorde is shown in Figure 4B.

We used Greedy Search as a heuristics solver for the TSP dataset. Greedy Search tries to identify the nearest neighbor for a given node and then travels to that node. Then it calculates the distance to remaining nodes and travels to the nearest node, while excluding first node. This approach often leads Greedy search to find sub-optimal tour for TSP. We observed that in our data set Greedy search gave the longest tour with a mean tour distance of 9.699. The non optimal tour of Greedy search is also visualized in Figure 4A, where Greedy search had a distance of 10.317, whereas Concorde found a solution with total distance of 7.771.

Finally, we verified the performance of GNN based approaches to predict TSP tours, namely Graph Convolution with Beamsearch and DPDP. Graph convolution with beam search is used as described by Joshi et al., on the 10,000 TSP dataset we have considered. We used a beam width of 1280 which is equal to the beam width described by Joshi et al. in [1]. By using this method we obtained predicted tours with mean tour distance of 7.869, which has an optimality gap of 1.399% with optimal solutions obtained by Concorde. One instance of TSP is visualized with the predicted optimal tour in Figure 4D, with a tour length of 7.908. This shows that Simple Graph Convolution with Beamsearch is not able to accurately predict the optimal tours. Finally, we used DPDP to predict tours for the TSP in our dataset. DPDP predicted tours with a mean distance of 7.765 which is very close to the optimal tour found by Concorde. The small optimality gap between DPDP and optimal solution is due to the rounding of predicted values.

Method	Mean Total Distance	Mean Optimality Gap	Runtime (s/graph)
Optimal (Concorde)	7.764	0.000%	0.060
Greedy Search	9.684	19.823%	0.037
Joshi et al. 2019-1280	7.897	1.681%	3.380
Joshi et al. 2019-10K	7.850	1.090%	25.72
Joshi et al. 2019-100K	7.784	0.260%	242.9
DPDP-1280	7.766	0.021%	0.329
DPDP-10K	7.765	0.007%	0.379
DPDP-100K	7.764	0.002%	0.833

Table 1. The Mean Total Distance and Mean Optimality Gap from the optimal solution for various methods to solve TSP. The values are averaged over 10,000 TSP of size 100 nodes. The optimal solution is generated by using Concorde, Greedy Search is performed by using Nearest Neighbor, Joshi et al. method uses Graph Convolution with Beamsearch with varying beam widths and DPDP is the DPDP based solution with varying beam widths

In [2], Kool et al. have used a beam width of 10K and 100K for finding the best possible tour for TSP. We felt this was an unfair assessment of the work by Joshi et al., as they had used beam width of size 1280. To assess the impact beam width we used the method described by Joshi et al. with a beam width of 10K and 100K. The analysis of impact of beam width on the tour length was carried out on a separate evaluation set of 10,000 graphs. On this new dataset there was an improvement in the predicted tour distance by increasing the beam width. For the Joshi et al. described method, the mean predicted tour length followed the order $7.897 > 7.850 > 7.784$ for the beam widths of size 1280, 10K and 100K respectively. However, it still falls short of the predicted tour by DPDP with mean predicted tour distance of $7.766 > 7.765 > 7.764$ for the beam widths of size 1280, 10K and 100K respectively. For this evaluation set of 10,000 graphs, the Concorde solver (optimal) gave a mean tour length of 7.764 and the greedy search gave mean tour length of 9.684. We have further calculated the runtimes for each of the methods, primarily to see how the time requirement scales with increasing the beam width. For the Joshi et al. method, increasing the beam width becomes prohibitively expensive while providing incremental gains in reducing the optimality gap. This can be viewed as a justification of the original work's selection of smaller beam width of 1280. The runtimes reported for the DPDP methods also utilize optimized sparse solvers, which plays a role in the increased performance. In case of DPDP, the nominal scaling of runtimes allows use of much larger beams and as a result reduces the optimality gap. It is noteworthy that under the constraints of same beam widths, the DPDP method clearly outperforms the Joshi et al. method. In fact, even with the small beam width in consideration, 1280, DPDP outperforms the latter with a beam width of 100K.

6 Conclusions

In conclusion, through our studies we found out that Graph Convolution based approaches are a considerable improvement over other heuristic based approaches. The recent work in this field is aimed at coupling multiple approaches together to find the optimal tours of TSP. Kool et al. [2] have successfully integrated the Graph Convolution and Beam search based TSP solver with Dynamic Programming to predict tours which are almost identical to the tours obtained by optimal solver, Concorde.

We can further analyze the behavior of the GNN based solvers by conducting benchmark tests on larger graphs, which have been largely untested to our best knowledge. The behavior would be dependent on tunable hyperparameters such as the batch size and beam size, which can be analyzed in a scalability test. The GNN-based approaches can be further improved upon by incorporating sparse graph neural networks for better scaling with larger graphs. Additionally, the scoring policy in the beams of DPDP is an ad hoc approach. This can be replaced with a more efficient scoring policy to find more optimal paths. These works can be expanded further by exploring additional datasets to understand how well these approaches might generalize. This may include random graphs of varying types or graphs sampled from existing real world problems, such as social graphs or transportation graphs.

7 Division of Work

Justin

1. Kool et al. 2019 [7] and Joshi et al. [1] model testing testing, verification and modification.
2. Deep Policy Dynamic Programming [2] mixed implementation
3. Data loading and plotting utility functions
4. Testing tutorials and scripts
5. Github environment control and continuous integration resolution
6. Main Report Contributions: Abstract, Literature Review, Baseline Selection, Contribution, Baseline Implementation, Experiments, Results, References, Figure 1 & 4, Table 1

Prakarsh

1. Joshi et al. [1] model testing, verification and modification
2. Method for TSP generation. Added functionality to generate TSP of any number of nodes in 2 dimensions. Added functionality to generate nodes from multiple probability distributions
3. Developed our version of Graph Convolution Network based on Joshi et al. work.
4. Reworked beamsearch algorithm for finding optimal tour on our heatmaps
5. Runtime analysis of Graph Convolution Network
6. Verified performance of Graph Convolution and beam search approach to find the optimal tour
7. Main Report Contributions: model description - Graph Convolution Network, Baseline selection, Baseline implementation, Results. Table 1, Figure 2, Figure 3 and Figure 4, References

Everett

1. Experimental implementations based on Kool et al. 2019
2. Appendix Model description - Some additional contributions to model extensions

Chaitanya

1. Concorde TSP model testing, verification, and integration
2. Greedy Search heuristic development, testing, and verification

3. Dataloader for input compatibility with Deep Graph Library
4. README updates, pushed code to Github
5. Main Report Contributions: Abstract, Figures on Concorde and Greedy Search, Literature Review, Baseline Implementation, References, edits for all sections

Code can be found at our Github [2]

8 Appendix

Model Architecture - Graph Attention Network

Let π be a permutation of $[|V|]$. π can be considered a closed tour on G if we assume that $\pi(0)$ and $\pi(-1)$ are connected.

Kool *et al.* use an attention based encoder-decoder model to iteratively and greedily find a minimum weight closed tour in G represented by π. To aid them in this task, they model a probability distribution over π given some problem instance s paramaterized by θ. That is, they find

$$p_\theta(\pi|s) = \prod_{t=1}^{n} p_\theta(\pi_t|s, \pi_{1:t})$$

This probability distribution is modeled by the encoder. The decoder than takes the embedding produced by the encoder and iteratively produces a candidate π.

Encoder

- let G be a Euclidean graph.
- let $n = |V|$ be the number of vertices in G.
- let $B \in \mathbb{N}$ be the batch size.
- let $d_x \in \mathbb{N}$ be the dimension of a vertex in G.
- let $d_h \in \mathbb{N}$ be the dimension of the embeddings.
- let $d_k = d_q \in \mathbb{N}$ be the dimension of the attention quarries and keys.
- let $d_v \in \mathbb{N}$ be the dimension of the attention unit values.
- let $M \in \mathbb{N}$ be the number of attention heads.
- let $N_h \in \mathbb{N}$ be the number of hidden units in the linear layer.
- let $x_i \in \mathbb{R}^{d_x}$ be a vertex in G.
- let $X \in \mathbb{R}^{(B \times n \times d_x)}$ be the design matrix.
- let $H^{(l)} \in \mathbb{R}^{(B \times n \times d_h)}$ be batch of hidden state matrices at time t.
- let $L \in \mathbb{N}$ be the total number of layers.
- let $W_1^{(l)} \in \mathbb{R}^{(d_h \times N_h)}$, $W_2^{(l)} \in \mathbb{R}^{(N_h \times d_h)}$, $b_1^{(l)} \in \mathbb{R}^{(1 \times N_h)}$, and $b_2^{(l)} \in \mathbb{R}^{(1 \times d_h)}$ be the set of parameters for the linear layer at layer $l \in [L]$.
- let $W_K^{(l)} \in \mathbb{R}^{(M \times d_h \times d_k)}$ be the key matrix in layer l.
- let $W_Q^{(l,m)} \in \mathbb{R}^{(M \times d_h \times d_q)}$ be the query matrix in layer l.
- let $W_V^{(l)} \in \mathbb{R}^{(M \times d_h \times d_v)}$ be the value matrix in layer l.
- let $W_O^{(l)} \in \mathbb{R}^{(M \times d_v \times d_h)}$ be the output matrix in layer l.
- let $\gamma^{(l)}$ be the $(n \times d_h)$ matrix of multiplicative batch norm weights in layer l.
- let $\beta^{(l)}$ be the $(n \times d_h)$ matrix of additive batch norm weights in layer l.

[2] https://github.com/salujajustin/GNN-for-CO

The encoder is defined by

$$H^{(0)} := XW_x + b_x$$
$$\hat{H}^{(l)} := BN^{(l)}\left(H^{(l)} + ATTN^{(l)}(H^{(l)})\right)$$
$$H^{(l+1)} := BN^{(l)}\left(\hat{H}^{(l)} + LL^{(l)}(\hat{H}^{(l)})\right)$$
$$\overline{H_{jk}}^{(L)} := \frac{1}{n}\sum_{i\in[n]} H^{(L)}_{jik}$$

We further specify $ATTN, LL$, and BN below.

Attention Unit: $ATTN(H)$

- let $K := HW_K$ be the $(B \times M \times n \times d_k)$ tensor of keys.
- let $Q := HW_Q$ be the $(B \times M \times n \times d_q)$ tensor of queries.
- let $V := HW_V$ be the $(B \times M \times n \times d_v)$ tensor of values.
- let $U := \frac{1}{\sqrt{d_k}}QK^T$ be the $(B \times M \times n \times n)$ compatibility tensor. note that each entry in $U_{bmij} = \frac{q_i^T k_j}{\sqrt{d_k}}$ where $q_i := Q_{bmi}$ and $k_j := K_{bmj}$.
- let $A_{bmij} := \frac{exp(U_{bmij})}{\sum_k exp(U_{bmik})}$ be an element of the $(B \times M \times n \times n)$ attention weight tensor. Note because this is the Softmax function, $\sum_i A_{bmij} = 1$.
- let $H' := AV$ be an intermediate $(B \times M \times n \times d_v)$ tensor.
- then $H^{ATTN}_{bij} := \sum_m (H'W_O)_{bmij}$ is an element of the $(B \times n \times d_h)$ output tensor from the attention unit.

Linear Layer: LL

$$LL^{(l)}(H^{(l)}) := ReLU\left(H^{(l)}W_1^{(l)} + b_1^{(l)}\right)W_2^{(l)} + b_2^{(l)}$$

Batch Norm: $BN(H)$

- let $E_{ij} := \frac{1}{B}\sum_{b\in[B]} H_{bij}$ be $(n \times d_h)$ expected value matrix over the batch B. Note that during inference E is replaced by a stored value.
- let $Var_{ij} := \frac{1}{B}\sum_{b\in B}(H_{bij} - E_{ij})^2$ be the $(n \times d_h)$ variance matrix over the batch B. Note that during inference Var is replaced by a stored value.
- let $\epsilon > 0$.
- then $H_{BN} := \left((H - E) \oslash \sqrt{Var + \epsilon}\right) \odot \gamma + \beta$ is the $(B \times n \times d_h)$ tensor output from a batch norm layer.

Decoder

The decoder has a somewhat similar mechanism to the encoder. It also uses an M-head attention. However, because the decoder determines the closed tour π iteratively, it has a limit context. To define this context, first we must define $h_i \in \mathbb{R}^(B \times 1 \times d_h)$ as the value in the ith row of H. Additionally let Π be the set of B paths corresponding to each element of the batch. Thus

$$h_i^{(L)} := H^{(L)}[:, i, :]$$

- let $v^1, v^f \in \mathbb{R}^{1\times 1\times d_h}$ be trainable initialization parameters

Then, we can define our context.

$$h^{(0)}_{(c)} := \overline{H}^{(L)} \oplus h^{(L)}_{\Pi_{t-1}} \oplus h^{(L)}_{\Pi_1}$$
$$h^{(t)}_{(c)} := \overline{h}^{(L)} \oplus V^1 \oplus V^f$$

where V^1 and V^f are the result of duplicating v^1, v^f B times and Where $\oplus$ denotes vector concatenation. This $h_{(c)}^{(t)}$ holds the necessary graph context to make inference, since for each element of the batch, we can only add new vertices to a single node $\pi_{t-1}^{(b)}$.

Now, we have another $M = 8$ head attention layer, except now $h_{(c)}^{(t)}$ is a $d_h + d_h + d_h$ dimensional vector. Therefore, we will have to modify the dimensions associated with W_Q.

Lastly, The output layer is defined as follows

- let $Q = h_{(c)} W_Q$ be an $(B \times 1 \times d_k)$ tensor of query values associated with c.
- let $K = HW_K$ be the $(B \times n \times d_k)$ tensor of key values.
- let $K' = MASK(HW_K)$ be the masked set of K values. Where MASK removes the rows associated with any node that is already included in $\pi^{(b)}(1 : t-1)$
- let $U = C * tanh\left(\frac{QK^T}{\sqrt{d_k}}\right)$ is the $(B \times 1 \times n)$ tensor of compatibilities.
- Finally, we have $P =$SoftMax(U) such that $P[b, i] = p_\theta(\pi_t^b = i | s, \pi_{1:t-1}^b)$

Loss and training

Similar to most other seq-to-seq tasks, once the probability distribution has been computed, we must find a way to select a sequence and compute a loss. Kool *et al.* opt to define $\mathcal{L}(\theta, s) := \mathbb{E}_{p_\theta(\pi|s)}[w(\pi)]$ where w is the euclidean weight function of the closed tour π.

Kool then uses the REINFORCE gradient estimator [20] with a baseline b(s). This is defined as:

$$\nabla \mathcal{L}(\theta|s) := \mathbb{E}_{p_\theta(\pi|s)}\left[(w(\pi) - b(s))\nabla \log p_\theta(\pi|s)\right]$$

here b(s) is a baseline with which we compare the performance of our policy. Kool *et al.* choose to define b(s) as the output of the greedy closed tour produced by the best model thus far (up to the current epoch). A t-test with $\alpha = 0.05$ is used to determine if a new model is significantly better than the prior best model. Here $w(\pi)$ is the weight of the closed tour π obtained by randomly sampling from the probability distribution. This is then averaged over the batch to give us the expectation. Lastly, The original paper uses the Adam optimizer to perform the parameter update step.

References

[1] Chaitanya K. Joshi, Thomas Laurent, and Xavier Bresson. An efficient graph convolutional network technique for the travelling salesman problem, 2019.

[2] Wouter Kool, Herke van Hoof, Joaquim Gromicho, and Max Welling. Deep policy dynamic programming for vehicle routing problems, 2021.

[3] Chaitanya K Joshi, Quentin Cappart, Louis-Martin Rousseau, Thomas Laurent, and Xavier Bresson. Learning tsp requires rethinking generalization. *arXiv preprint arXiv:2006.07054*, 2020.

[4] Marcelo Prates, Pedro HC Avelar, Henrique Lemos, Luis C Lamb, and Moshe Y Vardi. Learning to solve np-complete problems: A graph neural network for decision tsp. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 4731–4738, 2019.

[5] Jie Zhou, Ganqu Cui, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, Lifeng Wang, Changcheng Li, and Maosong Sun. Graph neural networks: A review of methods and applications, 2019.

[6] Zhuwen Li, Qifeng Chen, and Vladlen Koltun. Combinatorial optimization with graph convolutional networks and guided tree search. *arXiv preprint arXiv:1810.10659*, 2018.

[7] Max Welling Wouter Kool, Herke van Hoof. Attention, learn to solve routing problems! *arXiv preprint arXiv:1803.08475*, 2018.

[8] David Duvenaud, Dougal Maclaurin, Jorge Aguilera-Iparraguirre, Rafael Gómez-Bombarelli, Timothy Hirzel, Alán Aspuru-Guzik, and Ryan P. Adams. Convolutional networks on graphs for learning molecular fingerprints, 2015.

[9] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph Attention Networks. *International Conference on Learning Representations*, 2018.

[10] William L. Hamilton, Rex Ying, and Jure Leskovec. Inductive representation learning on large graphs, 2018.

[11] Ines Chami, Sami Abu-El-Haija, Bryan Perozzi, Christopher Ré, and Kevin Murphy. Machine learning on graphs: A model and comprehensive taxonomy, 2021.

[12] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and Philip S. Yu. A comprehensive survey on graph neural networks. *IEEE Transactions on Neural Networks and Learning Systems*, 32(1):4–24, Jan 2021.

[13] Yoshua Bengio, Andrea Lodi, and Antoine Prouvost. Machine learning for combinatorial optimization: a methodological tour d'horizon, 2020.

[14] Andrea Lodi and Giulia Zarpellon. On learning and branching: a survey. *Top*, 25(2):207–236, 2017.

[15] Yunhao Yang and Andrew Whinston. A survey on reinforcement learning for combinatorial optimization, 2020.

[16] Nina Mazyavkina, Sergey Sviridov, Sergei Ivanov, and Evgeny Burnaev. Reinforcement learning for combinatorial optimization: A survey, 2020.

[17] Natalia Vesselinova, Rebecca Steinert, Daniel F. Perez-Ramirez, and Magnus Boman. Learning combinatorial optimization on graphs: A survey with applications to networking. *IEEE Access*, 8:120388–120416, 2020.

[18] Quentin Cappart, Didier Chételat, Elias Khalil, Andrea Lodi, Christopher Morris, and Petar Veličković. Combinatorial optimization and reasoning with graph neural networks, 2021.

[19] Matthew Lamm Benedikt Bünz. Graph neural networks and boolean satisfiability. *arXiv preprint arXiv:1702.03592*, 2017.

[20] Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning., 1992.