Decision Transformer-based Traveling Salesman Tour Generation

by

Daniel S. Liu

B.S., Mathematics and Electrical Engineering and Computer Science,
Massachusetts Institute of Technology (2022)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2023

Authored By:	Daniel S. Liu Department of Electrical Engineering and Computer Science May 12, 2023
Certified by:	Hamsa Balakrishnan Professor, Aeronautics and Astronautics Thesis Supervisor
Accepted by:	Katrina LaCurts Chair, Master of Engineering Thesis Committee

Decision Transformer-based Traveling Salesman Tour Generation

by

Daniel S. Liu

Submitted to the Department of Electrical Engineering and Computer Science
on May 12, 2023, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

With the surge of new machine learning methods, research in classic problems like the Traveling Salesman Problem (TSP) is receiving a resurgence of popularity. One of the biggest goals in this renewed interest is to create a model that can not only outperform state-of-the-art heuristic solvers in speed for trivial sizes, but also generalize to larger TSP instances that are currently intractable. In this thesis we approach the TSP with the Decision Transformer, a transformer-based architecture transforming reinforcement learning environments into transformer-compatible sequence-modeling problems. By modeling a TSP instance as an graph-based environment with states and actions, we can input partial tours into the Decision Transformer to infer the next best action in an autoregressive fashion. With the power of the transformer, we take the first step in making headway on the issue of generalization where past models have failed.

Thesis Supervisor: Hamsa Balakrishnan
Title: Professor, Aeronautics and Astronautics

Research was sponsored by the United States Air Force Research Laboratory and the United States Air Force Artificial Intelligence Accelerator and was accomplished under Cooperative Agreement Number FA8750-19-2-1000. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the United States Air Force or the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notion herein.

Acknowledgments

I would like to acknowledge the MIT EECS department for giving me this opportunity for research and the DINaMO Group for funding this research. I'd also like to thank Prof. Hamsa Balakrishnan and Siddharth Nayak for their continued instruction and guidance, without which my progress on this research and in the field of machine learning in general would have been greatly diminished. I'd also like to thank Janice Yang, Steffi Ong, and my family for their continuous support during the process.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

The “Traveling Salesman Problem” (TSP) is a classic operations research problem first formulated in 1930 yet still studied today. A traveling salesman is given the task of selling merchandise in several cities, and needs to schedule a tour to visit each city exactly once, before returning back where they began. How should he schedule the tour to minimize the amount of distance he has to travel? Despite its simple appearance, this problem is NP-hard, making larger instances exponentially harder and eventually intractable to solve. This issue is what machine learning techniques aim to solve. As model inference typically takes time linear to the input size to the model, solutions found via machine learning models in comparison can scale arbitrarily well, making even the task of finding approximate solutions appealing. The main hurdle is thus to create a machine learning model that can zero-shot generalize well enough. In fact, the traveling salesman problem is only one of many operations research problems of interest in the machine learning community; Hubbs et al. (2020) provides a gym for designing models for various kinds of operations research problems including the traveling salesman problem. Here, we focus our efforts entirely on TSP.

1.1 Decision Transformer

When solving a reinforcement learning (RL) problem, we are given three things: the state of the world, the set of actions available to us, and the resulting reward once

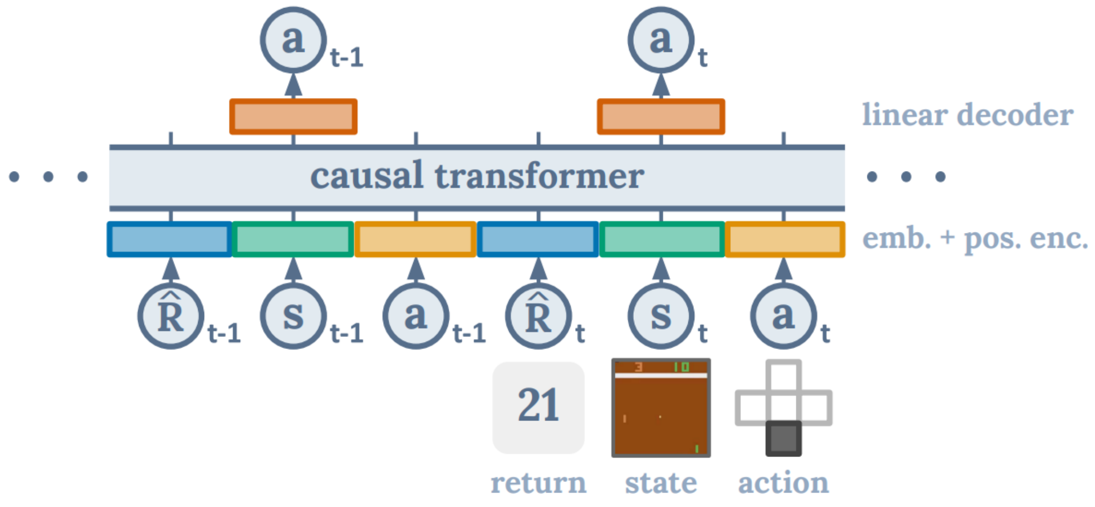


Figure 1-1: Decision Transformer architecture, as described in Chen et al. (2021) (used with permission).

we take an action. As we explore the space, we create a sequence of states, actions, and corresponding rewards. Standard solutions to reinforcement learning will try to propagate information on the amount of reward earned back to its causal actions to learn which actions were "good" and which were "bad". This is done via various methods like dynamic programming, temporal difference learning, or policy gradients. However, Chen et al. (2021) noted that since the reinforcement learning pipeline predicts the next action in a sequence, it may be possible to use sequence modelling methods common in natural language processing to do the same job. Their solution was the "Decision Transformer" (DT), a model based on the lauded transformer architecture (Vaswani et al., 2017) of natural language processing fame that processes state-action-rewards from a reinforcement learning environment in a sequence suitable for transformers.

The most natural context of the decision transformer is in *offline reinforcement learning*, where we are given a set of not necessarily optimal policies, and we wish to train a model to learn a best possible policy with these as reference. Thus, we are usually given as data a sequence of states, actions, and rewards from various policies; we are not allowed the liberty of exploring the environment ourselves.

Imagine we have a sequence of rewards, states, and actions:

$$r_1, s_1, a_1, r_2, s_2, a_2, \dots, r_T, s_T.$$

The idea of DT is to predict the next action a_T to take given this sequence. Structurally, this is done by utilizing the **transformer** architecture, an NLP-based model architecture renowned for its ability to make connections between distant elements of a sequence. This way, DT hopes to execute credit assignment better than traditional RL algorithms.

There is an important detail we need to clarify. As we currently set it up, we are feeding the model the reward r_t gathered at state s_t after taking action a_t. However, deciding what action to take given the reward we will gather prevents an ability to gain foresight. Thus, instead of giving rewards r_t in the sequence, we shall give **rewards-to-go** $\hat{R}_t$, defined as

$$\hat{R}_t = \sum_{t'=t}^{T} r_{t'}.$$

This is the amount of reward *remaining* to be gathered after action a_t is taken at state s_t. This way, the sequence knows from the beginning what "goal" return we wish to attain, and by the end it knows how much of that goal return we ended up achieving. This completes the paper's definition of the decision transformer's task:

Theorem 1 (Chen et al. (2021)) *Given a sequence of rewards-to-go, states, and actions*

$$\hat{R}_1, s_1, a_1, \hat{R}_2, s_2, a_2, \dots, \hat{R}_T, s_T,$$

the decision transformer predicts the action a_T taken at time T.

1.2 Graph Neural Networks

The data in TSP instances consists of cities and the cost it takes to get from one city to another. This is most naturally encapsulated in a **graph**, a mathematical object with nodes and edges between nodes.

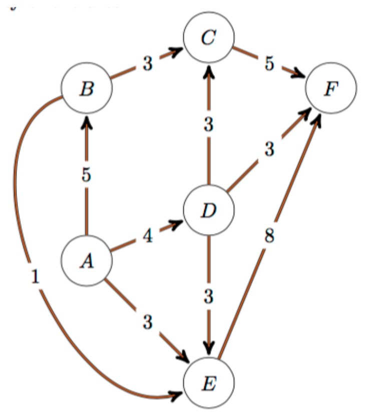


Figure 1-2: Example of a directed graph with letter node labels instead of numbers.

Definition 2 *A **graph** G with N nodes and E edges is an object with nodes which we label $1, \ldots, N$, as well as E edges e_{ij} (possibly with labels c_{ij}) connecting node i to node j, where $i, j \in [1, \ldots, N]$. If edge e_{ij} exists iff edge e_{ji} exists, the graph is known as **undirected**; otherwise, it is known as **directed**.*

However, working with graph data natively is not easy, as the structure is too abstract to easily be represented in the matrix-heavy framework of a machine learning model.

One could consider the **adjacency matrix** of the graph as a first pass on representing it. This is a square matrix where column or row indices correspond to node indices, and the element at row i and column j is the edge label c_{ij} if e_{ij} exists, and 0 otherwise. (If the graph edges are unlabelled, a label of 1 is used for edges that exist, and 0 otherwise.)

Example 1 *In the graph in Figure 1-2, the adjacency matrix M is*

$$M = \begin{bmatrix} 0 & 5 & 0 & 4 & 3 & 0 \\ 0 & 0 & 3 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 5 \\ 0 & 0 & 3 & 0 & 3 & 3 \\ 0 & 0 & 0 & 0 & 0 & 8 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}.$$

Here, the ith row/column corresponds to the ith letter of the alphabet.

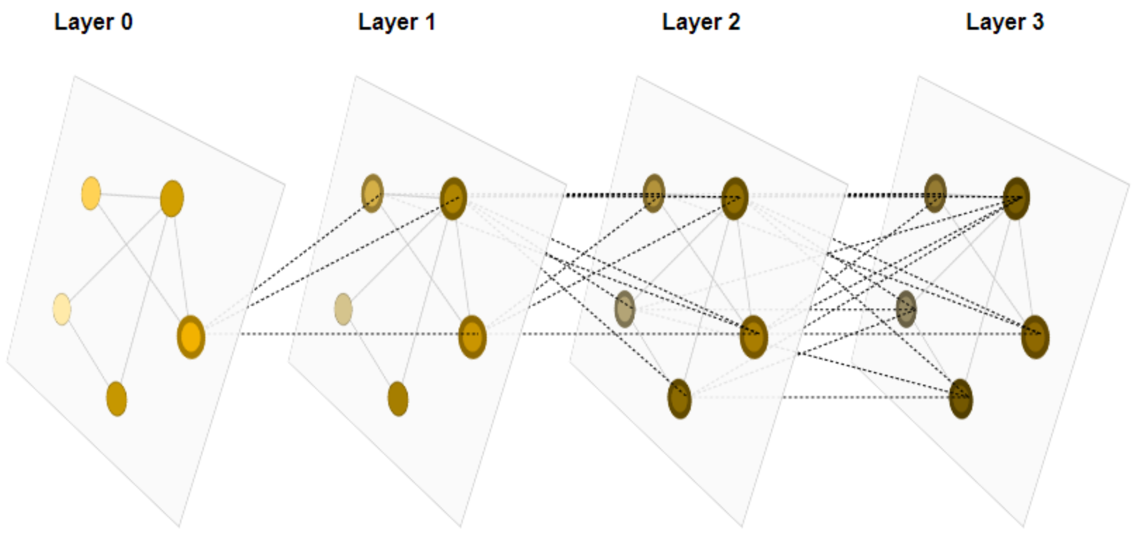


Figure 1-3: Message passing via three GNN layers. The dotted line indicates the trajectory of a message being passed from a node to the nodes in the next layer corresponding to the node's neighbors. Figure originally from Sanchez-Lengeling et al. (2021).

One could then imagine using classical machine learning methods for processing matrix data, such as convolutional neural networks. Unfortunately, since the order of the node labels is assigned arbitrary, the adjacency matrix has the property of row and column permutation invariancy, which CNNs unfortunately cannot deal with.

A theory based in machine learning principles that mirrors the functionality of the CNN is the **graph neural network**, or GNN. In parallel to how CNNs aggregate information between pixels within a certain locality, GNNs send messages along its edges to locally pass information via a **message passing protocol**.[1] These message-passing GNNs are also called **message-passing graph convolutional network** (MP-GCN) or just GCN for short. With more GCN layers, nodes in the graph are able to pass information to more distant nodes, mimicking the ability of CNNs to aggregate information across more distant pixels over several layers.

The design space for GNN layers is vast, with 13 distinct types of GNN surveyed in Dwivedi et al. (2020) alone. For ease of example, we shall provide the vanilla GCN

[1]GNN architectures that do not use message passing, e.g. Weisfeiler Lehman GNNs (Xu et al., 2018; Maron et al., 2019) exist but are beyond the scope of this thesis.

definition, a simplification of the graph convolutional network originally proposed by Kipf and Welling (2016).

Definition 3 (Adapted from Dwivedi et al. (2020)) *Let h_i^0 denote the node features of node i in the initial input layer. For each GCN layer $0 \leq \ell < L$ we define the output embedding*

$$h_i^{\ell+1} = \mathsf{ReLU}\left(U^\ell\left(\,\mathsf{Mean}_{j \in \mathcal{N}_i}\, h_j^\ell\right) + V^\ell\right).$$

Here, $\mathcal{N}_i$ denotes the set of neighbors of node i, and U^ℓ, V^ℓ are a trainable linear map and bias, respectively.

Given the final output embeddings h_i^L, we produce node, edge, and graph predictions respectively as follows:

$$\begin{aligned}\text{Node Prediction } i &= \mathsf{MLP}(h_i^L)\\ \text{Edge Prediction } e_{ij} &= \mathsf{MLP}(\mathsf{Concat}(h_i^L, h_j^L))\\ \text{Graph Predictions} &= \mathsf{MLP}\left(\mathsf{Mean}_{1\leq i\leq N}(h_i^L)\right)\end{aligned}$$

Chapter 2

Related Work

The predecessor of this thesis, Smith (2022), and also looked to find TSP tours with the Decision Transformer, but with a different method of defining the TSP environment states and actions. They adopt the standard method of defining the node the agent resides as the state, and the node the agent wishes to travel to as the action. However, due to the fact that this state action definition is so specific to the distribution of cities in the given instance, it does not produce an agent that can generalize to random city distributions and as a result there was no training improvement.

In other literature, there are three main approaches of using reinforcement learning for TSP: applying it on a fixed TSP instance, applying it as a step of a heuristic solution generator, or applying it over many distinct TSP instances.

The first class of approaches is the most straightforward application of reinforcement learning on TSP, where an agent explores a single TSP instance repeatedly until a permissible solution is found. Alipour et al. (2017) innovates on this fundamental approach by applying various optimizations after the reinforcement learning algorithm produces a tour.

Also worth mentioning are hybrid solvers imbuing a part of a heuristic solver with a reinforcement-learning based approach. da Costa et al. (2020) uses reinforcement learning to optimize choices in the 2-opt heuristic, a basic TSP heuristic based on swapping pairs of connections. Similarly, Zheng et al. (2021) investigates the possibility of using reinforcement learning techniques to aid in a single step of the Lin-Kernghan-Helsgaun (LKH) heuristic, currently one of the best combinatorial algorithms for finding TSP tours.

However, both of these main methods do not have the advantage of generalizing the agent's knowledge to different TSP instances without completely relearning. Thus, we are more interested in reinforcement learning approaches that span over many distinct TSP instances. Part of the challenge in this approach is determining a generalizable way of representing each TSP instance in a standardized format for model consumption. Miki et al. (2018) attacks this problem by representing TSP instances as images, and using a convolutional neural network model in both a reinforcement learning and a typical deep learning context. Meanwhile, Bello et al. (2016) takes the sequence of TSP coordinates as an input to an encoder-decoder recurrent neural network model and generates a permutation of the sequence as a tour.

With the rise of graph neural networks, utilizing their power in an obviously graph-natured problem like the TSP seems the best way to proceed. Song et al. (2021) makes this switch, using a graph neural network to calculate node embeddings, then autoregressively determining the next node to visit via a node-to-node attention mechanism.

An advantage of training a model over many TSP instances is the possibility of generalizing the solver to larger TSP instances unseen in training. Joshi et al. (2020) is one of the first to investigate this, applying both non-autoregressive and autoregressive attention-based GNN models to greater-sized TSP instances, but their generalization accuracy was limited. One of the main goals of this research is to achieve the power of generalization.

Chapter 3

The Model

In lieu of using the decision transformer, we view the TSP as a reinforcement learning problem. At any point in time on the tour, we consider the configuration of cities as well as the path we've taken so far to be the **state** of the environment, and the choice to travel to a city as the **action**. Furthermore, we associate the amount of distance (or in general, cost) it takes to travel to that city from the city we currently reside with a reward, defined in this context to be the negative distance between the cities.

The model we propose consists of two components, a GNN and a decision transformer. At a high level, we use the GNN as a tool to generate vector embeddings of graph elements that can be inputted into the decision transformer. These vector embeddings are used to create both the state embeddings and the action embeddings that are required for the decision transformer to work. Then, the decision transformer takes these embedding inputs, and outputs a predicted action embedding in the same embedding space as the actions supplied in the input. Finally, the best action to take is extracted by choosing the action out of all possible next moves most similar with the predicted action. During inference this is repetitively done in an autoregressive fashion until an entire path is constructed. Let's take a look at how this works in more detail.

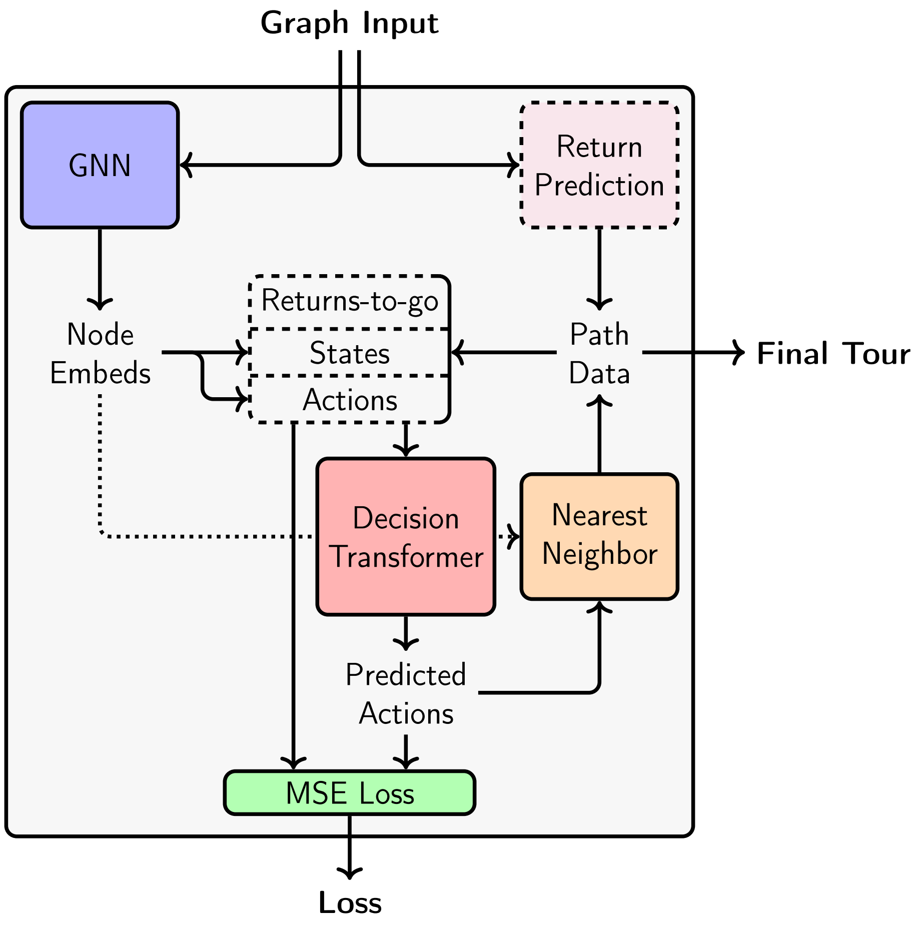
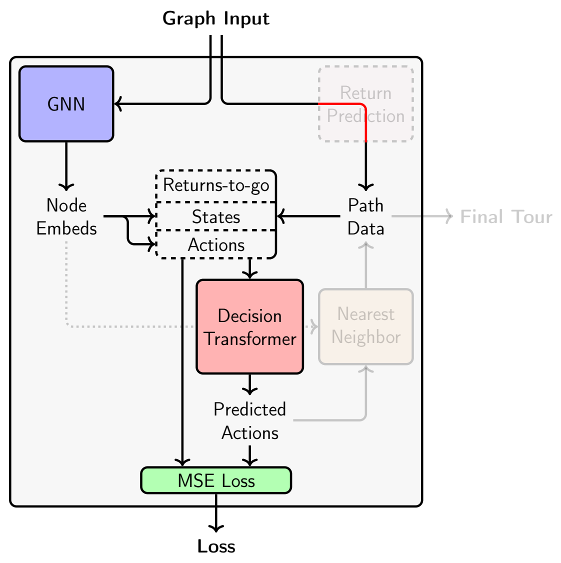


Figure 3-1: Decision transformer model containing a GNN input processing layer, an MSE loss function for training, and a nearest neighbor-based inference layer. Training directly computes a loss from a single pass-through, while inference involves autoregressively looping through the decision transformer until the path is entirely generated.



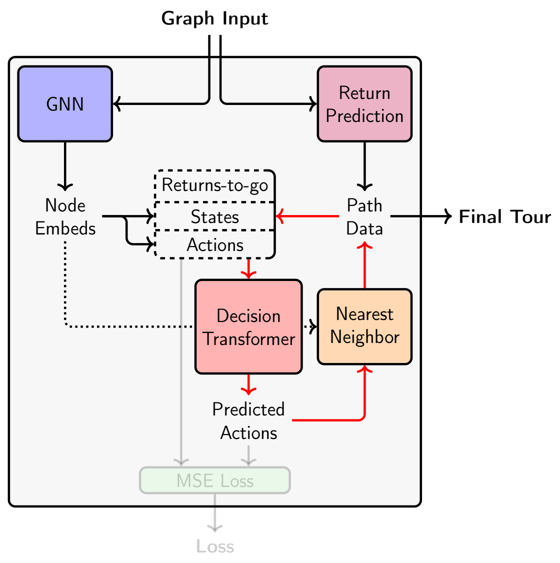


Figure 3-2: Training pathway (left). Inference pathway (right). Note (in red) the bypass of the "Return Prediction" module during training, and the autoregressive loop during inference.

3.1 The Data

We use the TSP data supplied by Dwivedi et al. (2020). This contains graphs of nodes randomly distributed in $[0,1]^2$, as well as a list of edges connecting each node to their k-nearest neighbor ($k = 25$) labelled with their Euclidean distances. The data contains 10,000 graphs for training, and 1,000 graphs each for validation and testing. To prepare the training and validation data for our model design in particular, we supplement them by generating several complete tours on each graph, represented by permutations of the graph's nodes in visiting order.

There are several ways we can generate these tours, each with a trade-off between generation speed and quality.

- **Randomized k-NN.** At each node, randomly choose a node in the k-nearest neighbors ($k = 25$) to travel to. If all nearest neighbors have already been visited, visit a random not yet visited node.

  This path generation is relatively poor in quality but very efficient to generate as we are already given the k-nearest neighbors by the data. In practice, this generation method is useful to determine the model's ability to improve on relatively poor offline examples.

- **Greedy with Optimization.**[2] A quick summary of the algorithm is as follows. Consider every node a path fragment of length 1. At each step, connect two path fragments with the smallest distance until two path fragments are left, then complete the tour with the shortest length possible. Afterwards, sweep through the tour a constant number of 2-opt cycles (Croes, 1958) to output a final tour.

  This path generation typically takes slightly longer to generate than the randomized k-NN method, but creates solutions of much higher quality. A full description of the algorithm can be found in Appendix A.

- **Concorde TSP Solver (Cook, 2003).** Generate the optimal path via the Concorde TSP Solver, currently regarded as the fastest algorithm that finds the optimal TSP tour.

  This path generation is significantly slower than the previous two sub-optimal tour generation methods, and is computationally intractable as a method of creating large volumes of offline tour data.

For our experiments we exclusively used the first two tour generation methods for training, mainly because generating optimal solutions on Concorde is computation-intensive, and from software incompatibility issues. Inference does not require the use of generated offline tours at all, as the model autoregressively generates the tour itself from scratch.

In addition to the TSP data from Dwivedi et al. (2020) we also create several more datasets that mirror this dataset, but with varying ranges of graph size. The dataset from Dwivedi et al. (2020), which we will call TSP, contains graphs with sizes ranging from 100 to 500. We also generate several more datasets, with properties listed below:

- TSP10: Contains graphs of size between 4 and 10.

[2]Code provided by the `tsp-solver2` Python package.

- TSP20: Contains graphs of size between 5 and 20.
- TSP50: Contains graphs of size between 12 and 50.
- TSP100: Contains graphs of size between 25 and 100.
- TSP200: Contains graphs of size between 50 and 200.
- TSP10f: (short for TSP10 fixed) Contains graphs of size exactly 10.

It is worth noting that for both training and inference, the tour is sliced into a contiguous subpath of context length K due to a limitation of transformers. Transformer attention layer computations scale in $O(n^2)$ which becomes intractable for arbitrarily long context lengths; see e.g. Dai et al. (2019).

3.2 The GNN Component

The design of the GNN component consists of two halves. As mentioned in Section 1.2, a GNN takes as input a graph, and outputs an embedding for each node and edge, as well as an embedding for the entire graph. After obtaining GNN embeddings, we then design a scheme to convert them into inputs suitable for the DT.

3.2.1 GNN Designs

We choose a fixed embedding dimension size of 64 for all GNN designs. Since node positions start out in $\mathbb{R}^2$, we first convert all positions to $\mathbb{R}^{64}$ through a fixed linear map.

Now let's describe the several design choices we tried for this GNN itself. For all of these designs, the graph itself consists of the union of all edges connecting each node to its 25-nearest neighbors.

- **Basic GCN.** We stack three layers of graph convolutional networks (GCN) as defined in Definition 3, with normalization layers after each GCN layer. Each normalization layer normalizes the node embeddings over the nodes of the graph to have mean 0 and variance 1, followed by a learnable scale factor and bias, with the exception of the last norm layer, which only has a learnable scale factor with no bias.

- **Deep GCN.** In order to increase the expressivity of the GNN, we can increase the number of GCN layers. However, it is well documented (Li et al., 2018) that GNNs quickly become ineffective with too many GCN layers due to vanishing gradients. Li et al. (2019) showed that if deep GNNs add residual (skip) connections, the same innovations that ResNet (He et al., 2015) made for deep neural networks, vanishing gradients can be avoided.

  We design our deep GNN to have residual blocks with two GCNs each, incorporating the skip connection from the input to the output of the residual block and aggregating the outputs via addition. In addition, we add normalization layers as defined in the previous section as well as ReLU's in the preactivation style, described in He et al. (2016) to be the most effective way to organize the layers within a residual block in ResNet.

- **Custom GNN.** The previous GNNs do not take into account the actual numerical distances between nodes, which could be suboptimal. Thus, we also try a custom-defined GNN that does take these distances directly in its message-passing scheme.

Definition 4 (Custom Message Passing Scheme) *Let h_i^0 denote the node features of node i in the initial input layer, and e_{ij} the distance between nodes i and j. For each GNN layer $0 \leq \ell < L$ we define the output embedding*

$$h_i^{\ell+1} = \mathsf{ReLU}\left(A^\ell\left(\mathsf{Mean}_{j\in\mathcal{N}_i}\, h_j^\ell\right) + B^\ell\left(\mathsf{Mean}_{j\in\mathcal{N}_i}\, e_{ij}^\ell\right) + C^\ell\right).$$

Here, $\mathcal{N}_i$ denotes the set of neighbors of node i, and A^ℓ, B^ℓ, and C^ℓ are two trainable linear maps and a bias, respectively.

This custom scheme was implemented in a normal three-layer GNN similar to the Basic GNN described above.

3.2.2 State-Action Embedding Schemes

After obtaining embeddings for the various components of a graph, we now define how to combine them into inputs suitable for the decision transformer to parse. This includes both the state of a TSP instance, as well as the action taken at that state, both as vector representations.

In order to capture the state of a partial TSP tour, fundamentally we must include the a representation of the list of nodes yet to be visited, as well as the tour's starting node and the node we currently reside, all in a conglomerated state vector. Note that other than the starting node, since we never visit a node twice, we can safely ignore all nodes we've already passed. For a redundancy of information we still choose to create an entire graph representation as well. We decide to represent each piece as follows:

- **Graph Representation.** We use the basic graph representation we introduced earlier, but without the MLP.

$$\mathbf{h}_G = \mathsf{Mean}_{1 \le i \le N}(h_i^L).$$

- **Unvisited Nodes Representation.** We consider a representation similar to the graph representation. Letting U_t be the set of nodes yet to be visited at timestep t, we get

$$\mathbf{h}_{U_t} = \mathsf{Mean}_{i \in U_t}(h_i^L).$$

- **Starting and Current Node Representation.** Letting n_i be the node visited at timestep i, the starting node representation is then simply

$$\mathbf{h}_0 = h_{n_0}^L,$$

  while the current node representation is

$$\mathbf{h}_t = h_{n_t}^L.$$

Together, we may create a conglomerate representation of the TSP state with a simple multi-layer perceptron.

- **State Representation.** Given the state of a partial TSP tour, we associate to it the embedding

$$\mathsf{TSP\ State}_t := \mathsf{MLP}(\mathsf{Concat}(\mathbf{h}_G, \mathbf{h}_{U_t}, \mathbf{h}_0, \mathbf{h}_t))$$

  which we denote in short by s_t. This MLP was chosen to be a simple two-layer feed forward neural network with output dimension 64, giving a state space of dimension 64.

Actions in the TSP setting correspond to single moves from a start node to a destination node. We create a single embedding by taking the signed difference between the end node embedding and the start node embedding.

- **Action Representation.** Given an action to move from city i to city j, we use the edge embedding

$$\mathsf{TSP\ Action}_t := e_{ij} = h_j^L - h_i^L$$

  which we denote in short by a_t. As we previously specified the GNN embedding dimension to be 64, the action space has dimension 64 as well.

This representation method does have a notable limitation. Both the state and action spaces are finite-dimensional. This limitation stems from the finite-dimensionality of the state and action space inputted into the decision transformer. However, as the arbitrary large graph size of the TSP instance naturally lies in a continuous infinite-dimensional state space $[0, 1]^\infty$, projecting this state information to the required finite-dimensional space is lossy. Further discussion about this issue can be found in Section 5.3.

3.2.3 A Control Representation

Because of this limitation, we also adopt a separate "control group" representation suitable only for TSP instances with exactly 10 cities. Here, states are given the common-sense definition of a direct enumeration of city coordinates:

- **State Representation (Control).** Let $\mathbf{p}_i$ be the position in $\mathbb{R}^2$ of the node during timestep i. Given the starting and current node locations $\mathbf{p}_0, \mathbf{p}_t \in \mathbb{R}^2$, as well as the sequence of nodes $\mathbf{p}_{t+1}, \ldots, \mathbf{p}_9 \in \mathbb{R}^2$ yet to be visited, the state is defined by $\mathsf{Concat}(\mathbf{p}_0, \mathbf{p}_t, \mathbf{p}_{t+1}, \ldots, \mathbf{p}_9)$. However, since as we proceed with tour generation the list of unvisited nodes shrinks, we pad this state by a vector **-1** of enough -1's to set the total length to be 22 (the max length obtainable by this state representation).

$$\mathsf{TSP\ State}_t := \mathsf{Concat}(\mathbf{p}_0, \mathbf{p}_t, \mathbf{p}_{t+1}, \ldots, \mathbf{p}_9, \mathbf{\text{-}1})$$

Meanwhile, the actions are represented identically to before, but taken directly from the positions of the source and destination nodes.

- **Action Representation (Control).** Let p_i be the position in $\mathbb{R}^2$ of node i. Then if at timestep t we choose to move from node i to node j, we define

$$\mathsf{TSP\ Action}_t := p_j - p_i.$$

Note that since these state action representations directly use node positions, this scheme does not utilize the GNN component at all. Thus, we can think of the control representation as a scheme that depends solely on the decision transformer to learn agent policies.

3.3 The Decision Transformer Component

The GNN provides us with a vector representation suitable for consumption by the decision transformer of both the TSP state and action at each timestep t of the tour.

In order to predict the next action to take (in the form of a prediction vector in action embedding space), we must finally supply the decision transformer with a sequence $\hat{R}_t$ of returns-to-go at each timestep t of the tour. This works differently depending on whether we're in training or inference mode.

- **Training.** We compute the returns-to-go for the entire path by computing the returns from each action (defined as the negative distance) and cumulatively summing:
$$\hat{R}_t = \sum_{t'=t}^{T} r_{t'}$$
  Then, we extract the returns-to-go relevant to the randomly chosen length K contiguous subpath of the tour.

- **Inference.** We initialize the inference with a "predicted" total return, estimated as precisely as possible. We choose to underestimate rather than overestimate, as giving the agent an underestimated unachievable return-to-go incentivizes it to try harder than giving it an easily achievable return-to-go. To this means we use the **1-tree TSP lower bound**, which is shown by Wolsey (1980) to be at worst 1.5 times shorter than the optimal TSP tour. Further details of this algorithm are outlined in Appendix B.

  After the initial return-to-go prediction, as per inference we autoregressively generate actions, and update the return-to-go with the reward obtained from the action.

In either case, we generate a return-to-go $\hat{R}_t$ for each timestep. The sequences $\{s_\tau\}_{\tau=t-K+1}^{t}$, $\{a_\tau\}_{\tau=t-K+1}^{t-1}$, and $\{\hat{R}_\tau\}_{\tau=t-K+1}^{t}$ (with a context length K) are then input to the decision transformer in the sequence

$$\hat{R}_{t-K+1}, s_{t-K+1}, a_{t-K+1}, \ldots, \hat{R}_t, s_t$$

in order to predict the next action $\hat{a}_t$.

In training, we propagate loss with a mean squared error loss function between $\hat{a}_t$ and the true a_t. However, in inference, this predicted action $\hat{a}_t$ will not correspond directly to a valid action to do at our current node. The actual action is chosen by calculating the valid action with the closest Euclidean distance to the predicted action $\hat{a}_t$. Explicitly, let i be the current node, and U_t be the set of nodes not yet visited at timestep t. Then the node j^* we visit next is determined by

$$j^* = \underset{j \in U_t}{\mathsf{argmin}} \|e_{ij} - \hat{a}_t\|_2$$

recalling that e_{ij} is the action embedding for traveling from city i to city j.

Chapter 4

Experiments and Results

4.1 Experimental Setup

All models are trained with stochastic gradient descent (Bottou and Bousquet, 2007) with a learning rate of 0.01, and a minimum squared error loss. Models were trained on one of the listed datasets mentioned in Section 3.1 for five epochs, or sometimes for twenty epochs if more training was needed. Based on the dataset and path generation chosen, the experiment represents one of several learning heuristics.

- **Large Datasets.** This includes TSP and TSP200, and improved performance represents the ability of the model to make good macroscopic optimizations.
- **Small Datasets.** This includes TSP10 and TSP10f, and improved performance represents the ability of the model to find the optimal solutions at the microscopic scale.
- **Randomized Tours.** This is the option of generating offline path data via the randomized k-NN algorithm. Improved performance represents the model's ability to learn from and surpass low quality path generation.
- **Greedy Tours.** This is the option of generating offline path data via a greedy method with optimization. Improved performance represents the model's ability to learn from and surpass relatively high quality path generation.

4.2 Results

Models were initially tested on the same dataset that they were trained on, with the intention of testing on datasets with larger graphs afterwards to test for generalizability.

The industry-standard metric of rating models is by how much longer in percentage the model solution is compared to a benchmark. Typically the benchmark is the optimal solution. We found it more instructive to set the benchmark to the generated path data for comparison for three reasons. First, bear in mind that the model only has *offline* paths generated by either a randomized k-NN or an optimized greedy algorithm as input for its training. Thus, it is natural to compare the model's performance against the performance of the paths it is given as training to determine whether the model has the power to not only learn the algorithm's path generation from scratch, but to even exceed it in path efficiency. Second, our model unfortunately was not competitive against state-of-the-art algorithms, so comparing its length to the optimal path length is less revealing. The optimized greedy algorithm given in the `tsp-solver` Python package generates visually acceptable solutions, so we believe comparing the model performance to the optimized greedy algorithm is a good enough first-order approximation of performance.

Thus, we instead measure performance by how much *shorter* in percentage the model solution's length is compared to the path generated for offline training. Here, a positive percentage indicates that the model's solution is shorter than the offline generated path, which indicates that the model is learning better movement than the data it was given. On the other hand, a negative percentage indicates that the model's solution was worse than the offline generated path, which indicates the model's failure to learn past the threshold of the data it was given. For completeness we also supply the absolute tour length of the model's paths and benchmark paths.

Model Type	Tour Type	Dataset	U	T	B	%U	%B
GCN	Random kNN	TSP	32.75	21.31	37.80	**34.9**	**43.6**
GCN	Greedy	TSP	31.44	21.09	13.00	32.9	−62.2
DeepGCN	Greedy	TSP	19.52	20.02	13.00	−2.6	−54.0
CustomGNN	Greedy	TSP	17.13	**15.99**	13.00	6.7	−23.0
GCN	Random kNN	TSP10	3.57	3.54	3.40	0.8	−4.1
GCN	Greedy	TSP10	3.58	3.59	2.46	−0.3	−45.9
DeepGCN	Greedy	TSP10	3.16	3.14	2.46	0.6	−27.6
CustomGNN	Greedy	TSP10	**2.72**	**2.72**	2.46	0.0	−10.6
Control	Greedy	TSP10f	3.47	3.28	2.96	5.5	−10.8

Table 4.1: Experimental results of the model as we vary the dataset, pre-generated tour type, and model type. Listed are the (U)ntrained model tour length, (T)rained model tour length, (B)enchmark (pre-generated tour) length, as well as the (%U) and (%B) percent improvement of the trained model length over the untrained model and benchmark length,respectively. Model was trained on five epochs; test data was averaged over 100 test trials.

From Table 4.1, we conclude the following:

- Out of all the experiments, the basic GCN model trained on randomized kNN tour data saw the greatest improvement over both the untrained model and the tour data, with a percentage improvement of 34.9% and 43.6% respectively. As a result, the majority of the remaining experiments focused on achieving the same performance improvement on the higher quality greedy tour data.

- The greedy tour data is much shorter than the random kNN tour data, with an average tour length of 13.00 compared to an average tour length of 37.80.

- The custom GNN design saw the best performance on the TSP dataset with greedy tour data, with an average untrained tour length of 17.13 and an average trained tour length of 15.99. However, this still fell 23.0% short of the average greedy tour data length of 13.00.

- The custom GNN design saw the best performance on the TSP10 dataset with greedy tour data as well, with a tour length of 2.72 on both the untrained and trained models.

- The trained model saw almost no improvement over the untrained model for the TSP10 datasets, and the margin of improvement decreased on the TSP dataset as the overall untrained performance improved.

Chapter 5

Analysis and Future Work

The model was able to not only mirror, but exceed the performance of the offline random k-NN path generation, producing paths on average 43.6% more efficient than the random k-NN paths. This provides evidence to the crucial ability for the model to take as training input solutions of a certain level of quality, and output solutions of higher quality. Ideally, if this property were to hold true in general, the model should be able to take in high quality solutions and learn to generate even higher quality solutions.

Contrary to this inductive reasoning, we observe an inability for the model to improve past the performance threshold set by the optimized greedy algorithm path generation, becoming "stuck" at a certain performance level even after further training. We can glean on the model's deficiencies by observing some example tours in Figure 5-1. It appears that the model is able to formulate well-formulated paths between nodes locally. Rarely are small clusters of nodes "forgotten", classified by a large jump to said cluster followed immediately by another large jump after the cluster is visited. However, it has trouble avoiding crosses between two sequentially distant sections of path, as well as the tendency to jump large distances between stretches of well-formulated paths. These factors combined results in a less optimal tour than the greedy algorithm.

Next we discuss the various attempts at breaking through this performance ceiling as listed in the findings of Chapter 4, as well as other potential factors that we did

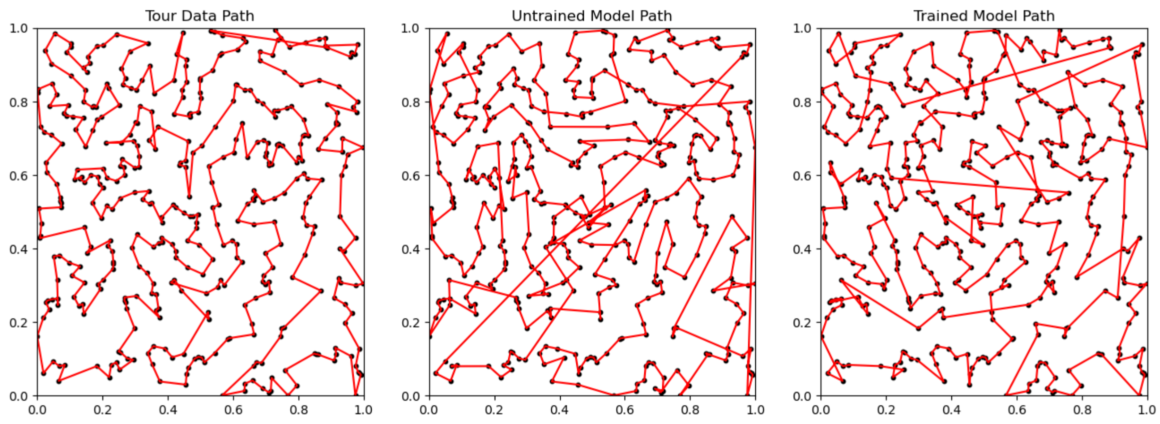


Figure 5-1: The greedy-based tour data, the untrained model's path, and the trained model's path on an example TSP instance for the CustomGNN model.

not experiment on.

5.1 Factors in Model Design

The main part of this model's design that could fundamentally affect the performance is the difference in logic path between training and inference. Training and validation loss is calculated through comparing the predicted action embeddings with the actual action embeddings of the pre-generated tour through a single pass-through of the decision transformer. On the other hand, the autoregressive prediction style of inference necessitates many recursive calls on the decision transformer, and the restriction of generating actions that move the agent to valid nodes necessitates an extra nearest neighbors component in the model not seen by training and validation. These differences could very well cause a discrepancy between training/validation performance and inference performance.

The naive method of reconciling this difference is to have training and validation follow the same logic path as inference, and calculate the negative log likelihood loss directly on the generated tour as a classification objective. There are two reasons why this idea could fail. First, computing a loss between the model's final tour output and the pre-generated training data tour is a further logical leap than comparing single actions between the output and the training data. One would expect that generating a loss in the action prediction process should guide the model in a more fine-grained way than a single tour-based loss at the end. Second, autoregressively generating a tour from scratch calls the decision transformer many more times than calling it to compare actions. Indeed, generating a tour from scratch to compute a single loss value necessitates the same number of calls to the decision transformer as there are nodes in the graph, while N calls to the decision transformer at random length K subsequences of the tour already produces $N \cdot K$ predicted actions to compute a loss to. For the sake of computational efficiency during training, the current model provides the best compromise

Future work on this issue could involve computing a loss from both comparing actions and comparing final tour results. There is plenty of precedence in applying loss functions in this way; in the field of knowledge distillation, Gou et al. (2020) trains student networks based on both a logit loss and a distillation loss, which closely parallels this model's predicted action-based loss and final tour loss, respectively. The issue of computational efficiency remains a hurdle this approach would need to overcome.

5.2 Factors in GNN Design

Another component that we thought could affect the efficacy of the model is the design of the GNN component. As a result, we experimented on three GNN designs as mentioned in Section 3.2.1. We iterated through several GNN designs in Section 3.2.1 to try to include as much information as possible for the GNN to consume and interpret. Section 4.2 shows that iterating the GNN design had an improvement in baseline performance for both the TSP dataset and TSP10 dataset. For example, between the GCN, DeepGCN, and CustomGNN designs on the TSP dataset, the untrained model achieved a steadily improving tour length of $31.44, 19.52$, and 17.13 respectively, while the trained model exhibited improving tour lengths of $21.09, 20.02$, and 15.99 respectively. Figure 5-2 shows an example of the performance of the GCN model versus the CustomGNN model on a TSP instance.

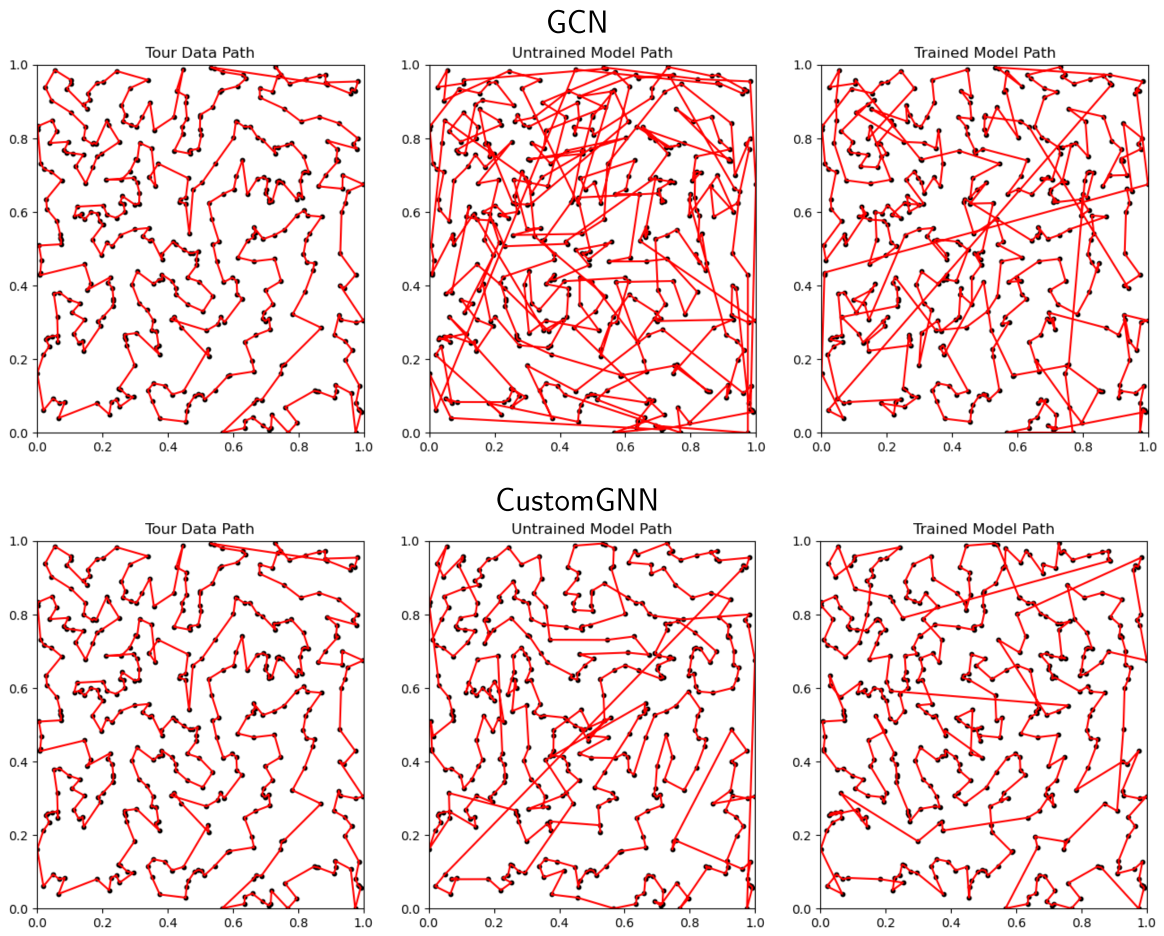
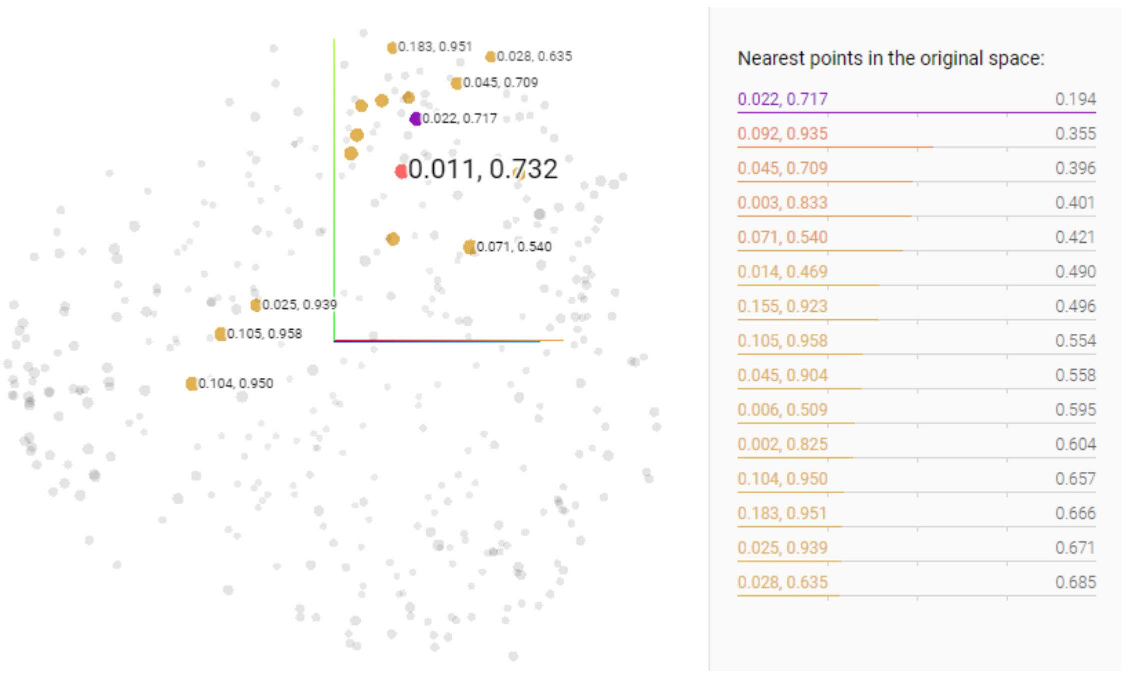


Figure 5-2: Comparison between the greedy-based tour data, the untrained model's path, and the trained model's path on an example TSP instance for both the basic GCN model and the CustomGNN model. Observe that through the use of CustomGNN, even the untrained tour appears better constructed than the GCN model's tours.



Nearest points in the original space:

0.022, 0.717	0.194
0.092, 0.935	0.355
0.045, 0.709	0.396
0.003, 0.833	0.401
0.071, 0.540	0.421
0.014, 0.469	0.490
0.155, 0.923	0.496
0.105, 0.958	0.554
0.045, 0.904	0.558
0.006, 0.509	0.595
0.002, 0.825	0.604
0.104, 0.950	0.657
0.183, 0.951	0.666
0.025, 0.939	0.671
0.028, 0.635	0.685

Figure 5-3: Principal component analysis of the GNN node embeddings of a randomly chosen graph. Shown is a randomly chosen node, as well as its list of nearest neighbors based on node embedding. Each node is also labelled with its original position in $[0,1]^2$, revealing that nodes which are considered nearest neighbors via node embedding are close in absolute position to the original node position as well. (Diagram generated by TensorBoard Embedding Projector.)

Unfortunately, changing the GNN design did not make any significant breakthrough in the performance of the model in relation to the tour data performance as all of the comparisons $\%B$ on the TSP dataset heavily leaned negative. However, there is reason to believe that the GNN is not necessarily the component that is causing learning issues. A principal component analysis (Figure 5-3) of the GNN embeddings of the three-layer GCN we implemented does give evidence that even this most basic iteration of the GNN produces node embeddings that correctly preserves relative distance.

Finally, it is interesting to note that although the design of the GNN was not able to allow the model to train better, it did affect how well the model performed untrained on observation. Perhaps this is due to better relative distance preservation of node embeddings. Future work should be done to better understand why one GNN design may improve the baseline untrained model performance more than another.

5.3 Factors of State-Action Embedding Schemes

The embedding scheme we propose in Section 3.2.2 came about as a compilation of all the information intuitively needed to produce quality TSP tours, but any embedding scheme could have been used. For example, a previous iteration of the embedding scheme we propose here was similar except we produce an embedding corresponding to the set of nodes we have *already visited* instead of the set of nodes we have yet to visit. There was no observed performance change under this modification, so it was omitted from the results.

There is reason to believe that the necessity of creating a state-action embedding scheme is a cause for the performance limitations. As mentioned briefly in Section 3.2.2, we must specify the state and action embeddings to exist in a finite dimensional state space $\mathcal{S} \cong \mathbb{R}^{64}$ and action space $\mathcal{A} \cong \mathbb{R}^{64}$, respectively. However, the state of a TSP instance includes the positions of an arbitrary number of nodes, which naturally lies in the space $[0,1]^\infty$. Bearing in mind the finite precision of floats, there is no injective map from $[0,1]^\infty$ to $\mathcal{S}$, with the implication that there will always exist some state representations in $\mathcal{S}$ that correspond to more than one configuration of nodes in $[0,1]^\infty$. In short, there is no way of perfectly representing a TSP state in a finite-dimensional space; there is always some amount of information loss.

Nevertheless, the current embedding scheme mainly utilizes node embedding averaging over a set of nodes to represent that set of nodes, which is particularly lossy. Future work should try to find a method of representation that, although is lossy, still acts as a good approximate representation of the original TSP state.

5.4 Lessons of the Control Experiment

As a final control experiment, we created a state-action embedding scheme that is taken directly from the positions of the nodes, assuming that the number of nodes is fixed at exactly 10. This control experiment sought to eliminate as much influence from the previous factors as possible, and isolate the decision transformer as the main component of the model to determine whether the decision transformer itself could be the limiting factor.

Indeed, with the use of node positions directly instead of GNN embeddings, we remove the GNN from the pipeline entirely. In addition, with the number of nodes set at exactly 10, we remove the non-unique representation concern of the TSP state. Thus, the concerns of neither Sections 5.2 and 5.3 should affect this model design, with the decision transformer essentially acting as the only learnable component.

Unfortunately, we see that even this simplified model was unable to break through a performance barrier and produce optimal solutions for the 10-node TSP. Thus, we conclude that either the design of the model is suspect as Section 5.1 describes, or the use of the decision transformer itself in the TSP is not sufficient.

Future work should be done to verify whether the limitations are due to the offline decision transformer-based model or due to something inherent in the use of reinforcement learning on distinct TSP instances itself. One proposal to achieve this could be training a model in an online style with direct reward responses from the environment after every agent action as opposed to the offline supervised model done here. It is worth noting though that this proposed online model is not a complete analogue of the offline model done here as there is no longer any pre-generated path data to supervise the model, so its success or failure does not necessarily correlate with the success or failure of the decision transformer.

Chapter 6

Conclusion

The decision transformer has some potential in offline reinforcement learning. Through the use of the a model powered by the decision transformer, we see its ability to generate TSP tours significantly shorter than the tours it is fed as training data in the case of low quality tour data. We also see the ability for it to formulate good tour segments given higher-quality tour data.

Unfortunately, due to high-level errors like crossings and large jumps between well-constructed paths, the decision transformer model does not output tours of better quality than its input tour data. We experimented with several measures to fix this issue, including changing the GNN specifications, changing the state-action embedding schemes, and even stripping the model down to the bare minimum of just the decision transformer component, but the model saw no significant improvement.

Nevertheless, the paths produced by the model appear a promising first step, and the model was able to learn and improve past its training data in at least one circumstance, confirming its ability to infer past its baseline training data performance. It remains as further investigation to refine the model to succeed in a higher-performance training environment for the traveling salesman problem.

Appendix A

`tsp-solver2`'s Greedy TSP Solver

Here is a full description of the optimized greedy algorithm implemented by the `tsp-solver2` Python package, used as the high-quality offline path generation for our model training data.

Algorithm 1 Greedy-based TSP solver, tour generation step

```
Require: $n \geq 3$, $V = \{v_i\}_{1 \leq i \leq n}$, $E = \{e_{ij}\}_{1 \leq i \neq j \leq n}$, $G = (V, E)$ symmetric
  $V_{\text{endpoints}} \leftarrow V$
  $E_{\text{tour}} \leftarrow \emptyset$
  while $|E_{\text{tour}}| < n - 1$ do
      $E_{\text{valid}} \leftarrow \{e_{ij} \in E - E_{\text{tour}} \mid i, j \in V_{\text{endpoints}}, e_{ij}$ does not form a cycle in $E_{\text{tour}}\}$
      $(a, b) \leftarrow \mathsf{argmin}_{a,b}\{e_{ab} \mid e_{ab} \in E_{\text{valid}}\}$
      $E_{\text{tour}} \leftarrow E_{\text{tour}} \cup \{e_{ab}\}$
      if $\deg_{E_{\text{tour}}}(a) = 2$ then
          $V_{\text{endpoints}} \leftarrow V_{\text{endpoints}} - \{a\}$
      end if
      if $\deg_{E_{\text{tour}}}(b) = 2$ then
          $V_{\text{endpoints}} \leftarrow V_{\text{endpoints}} - \{b\}$
      end if
  end while
  $\{s, t\} \leftarrow V_{\text{endpoints}}$
  $E_{\text{tour}} \leftarrow E_{\text{tour}} \cup \{e_{st}\}$
  return $E_{\text{tour}}$
```

In words, Algorithm 1 repeatedly adds the shortest edge that connects two degree ≤ 1 nodes ($V_{\text{endpoints}}$) such that the edge does not prematurely form a cycle, until one big path encompassing every node is created, for when the endpoints are joined to create a full tour.

Next, this tour is plugged into Algorithm 2 a constant number of times (default to 3) to further optimize it.

Algorithm 2 Greedy-based TSP solver, optimization step (2-opt)

```
Require: $n \geq 3$, $V = \{v_i\}_{1 \leq i \leq n}$, $E = \{e_{ij}\}_{1 \leq i \neq j \leq n}$, $G = (V, E)$ symmetric
Require: $E_{\text{tour}} \subset E$ is a cycle, $|E_{\text{tour}}| = n$
  $a \leftarrow 1$
  while $a \leq n - 2$ do
      $b \leftarrow a + 1$
      $c \leftarrow b + 1$
      while $c \leq n$ do
          if $c = n$ then
              $d \leftarrow 1$
          else
              $d \leftarrow c + 1$
          end if
          $\delta \leftarrow e_{ab} + e_{cd} - e_{ac} - e_{bd}$
          if $\delta > 0$ then
              $E_{\text{tour}} \leftarrow E_{\text{tour}} - \{e_{ab}, e_{cd}\}$
              $E_{\text{tour}} \leftarrow E_{\text{tour}} \cup \{e_{ac}, e_{bd}\}$
          end if
          $c \leftarrow c + 1$
      end while
      $a \leftarrow a + 1$
  end while
```

In words, Algorithm 2 loops through pairs of edges in the tour, and upon finding that the four nodes involved in these two edges have a shorter pair of edges that connect them, swaps the original edges for the shorter pair. This algorithm is known as 2-opt.

Appendix B

1-Tree TSP Lower Bound

More sophisticated lower bounds involving 1-trees exist such as the Held-Karp lower bound (Held and Karp, 1970), but for our purposes a very rough lower bound should suffice. Here we use a single 1-tree as a lower bound. Let G be an undirected graph with edge distances e_{ij} between nodes i and j. Let $G - v$ for a graph G and a vertex $v \in G$ be the subgraph induced by the vertices in G excluding v. Let MST be a standard minimum spanning tree algorithm such as Prim's or Kruskal's. Define $[n] := \{1, 2, \dots, n\}$.

Algorithm 3 Calculating a basic 1-tree lower bound for a generic TSP instance

```
Require: $n \geq 3$, $V = \{v_i\}_{1 \leq i \leq n}$, $E = \{e_{ij}\}_{1 \leq i \neq j \leq n}$, $G = (V, E)$ symmetric
  $a \leftarrow \mathsf{argmin}_{j \in [n]-\{1\}} \{e_{1j}\}$
  $b \leftarrow \mathsf{argmin}_{j \in [n]-\{1,a\}} \{e_{1j}\}$
  $T \leftarrow \mathsf{MST}(G - v_1)$
  return $e_{1a} + e_{1b} + \mathsf{len}(T)$
```

The proof that this is indeed a lower bound for the optimal TSP tour is simple. Note that v_1 must be entered and exited exactly once each in the optimal TSP tour; the distance traveled during these two actions must then be at least the sum $e_{1a} + e_{1b}$ of the distances from v_1 to the closest two nodes v_a, v_b. Next, delete node v_1, and observe that the optimal tour creates a path on the remaining nodes. Since a path is a tree, the distance traveled on this path must be at least the total length of the minimal spanning tree T of $G - v_1$, giving us a total lower bound of $e_{1a} + e_{1b} + \mathsf{len}(T)$.

Bibliography

M. M. Alipour, S. N. Razavi, M. R. F. Derakhshi, and M. A. Balafar. A hybrid algorithm using a genetic algorithm and multiagent reinforcement learning heuristic to solve the traveling salesman problem. *Neural Computing and Applications*, 30 (9):2935–2951, Feb. 2017. doi: 10.1007/s00521-017-2880-4. URL https://doi.org/10.1007/s00521-017-2880-4.

I. Bello, H. Pham, Q. V. Le, M. Norouzi, and S. Bengio. Neural combinatorial optimization with reinforcement learning, 2016.

L. Bottou and O. Bousquet. The tradeoffs of large scale learning. In J. Platt, D. Koller, Y. Singer, and S. Roweis, editors, *Advances in Neural Information Processing Systems*, volume 20. Curran Associates, Inc., 2007. URL https://proceedings.neurips.cc/paper_files/paper/2007/file/0d3180d672e08b4c5312dcdafdf6ef36-Paper.pdf.

L. Chen, K. Lu, A. Rajeswaran, K. Lee, A. Grover, M. Laskin, P. Abbeel, A. Srinivas, and I. Mordatch. Decision transformer: Reinforcement learning via sequence modeling, 2021.

W. J. Cook. Concorde tsp solver, 2003. URL https://www.math.uwaterloo.ca/tsp/concorde.html.

G. A. Croes. A method for solving traveling-salesman problems. *Operations Research*, 6(6):791–812, 1958. ISSN 0030364X, 15265463. URL http://www.jstor.org/stable/167074.

P. R. d. O. da Costa, J. Rhuggenaath, Y. Zhang, and A. Akcay. Learning 2-opt heuristics for the traveling salesman problem via deep reinforcement learning. In S. J. Pan and M. Sugiyama, editors, *Proceedings of The 12th Asian Conference on Machine Learning*, volume 129 of *Proceedings of Machine Learning Research*, pages 465–480. PMLR, 18–20 Nov 2020. URL https://proceedings.mlr.press/v129/costa20a.html.

Z. Dai, Z. Yang, Y. Yang, J. Carbonell, Q. V. Le, and R. Salakhutdinov. Transformer-xl: Attentive language models beyond a fixed-length context, 2019.

V. P. Dwivedi, C. K. Joshi, A. T. Luu, T. Laurent, Y. Bengio, and X. Bresson. Benchmarking graph neural networks, 2020.

J. Gou, B. Yu, S. J. Maybank, and D. Tao. Knowledge distillation: A survey. *International Journal of Computer Vision*, 2020. doi: 10.1007/s11263-021-01453-z.

K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition, 2015.

K. He, X. Zhang, S. Ren, and J. Sun. Identity mappings in deep residual networks, 2016.

M. Held and R. M. Karp. The traveling-salesman problem and minimum spanning trees. *Operations Research*, 18(6):1138–1162, Dec. 1970. doi: 10.1287/opre.18.6.1138. URL https://doi.org/10.1287/opre.18.6.1138.

C. D. Hubbs, H. D. Perez, O. Sarwar, N. V. Sahinidis, I. E. Grossmann, and J. M. Wassick. Or-gym: A reinforcement learning library for operations research problems, 2020.

C. K. Joshi, Q. Cappart, L.-M. Rousseau, and T. Laurent. Learning the travelling salesperson problem requires rethinking generalization. *Constraints*, 2020. doi: 10.4230/LIPIcs.CP.2021.33.

T. N. Kipf and M. Welling. Semi-supervised classification with graph convolutional networks, 2016.

G. Li, M. Müller, A. Thabet, and B. Ghanem. DeepGCNs: Can GCNs go as deep as CNNs?, 2019.

Q. Li, Z. Han, and X.-M. Wu. Deeper insights into graph convolutional networks for semi-supervised learning, 2018.

H. Maron, H. Ben-Hamu, H. Serviansky, and Y. Lipman. Provably powerful graph networks, 2019.

S. Miki, D. Yamamoto, and H. Ebara. Applying deep learning and reinforcement learning to traveling salesman problem. In *2018 International Conference on Computing, Electronics & Communications Engineering (iCCECE)*, pages 65–70, 2018. doi: 10.1109/iCCECOME.2018.8659266.

B. Sanchez-Lengeling, E. Reif, A. Pearce, and A. B. Wiltschko. A gentle introduction to graph neural networks. *Distill*, 2021. doi: 10.23915/distill.00033. https://distill.pub/2021/gnn-intro.

C. Smith. Attention-based learning for combinatorial optimization. Master's thesis, Massachusetts Institute of Technology, 2022.

Y. Song, J. Davis, E. Duthie, and C. Wu. Solving the traveling salesperson problem with deep reinforcement learning on amazon sagemaker, Sep 2021. URL https://aws.amazon.com/blogs/opensource/solving-the-traveling-salesperson-problem-with-deep-reinforcement-learning-on-amazon-sagemaker/.

A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin. Attention is all you need, 2017.

L. A. Wolsey. Heuristic analysis, linear programming and branch and bound. In V. J. Rayward-Smith, editor, *Combinatorial Optimization II*, pages 121–134. Springer Berlin Heidelberg, Berlin, Heidelberg, 1980. ISBN 978-3-642-00804-7. doi: 10.1007/BFb0120913. URL https://doi.org/10.1007/BFb0120913.

K. Xu, W. Hu, J. Leskovec, and S. Jegelka. How powerful are graph neural networks?, 2018.

J. Zheng, K. He, J. Zhou, Y. Jin, and C.-M. Li. Combining reinforcement learning with Lin-Kernighan-Helsgaun algorithm for the traveling salesman problem. *Proceedings of the AAAI Conference on Artificial Intelligence*, 35(14):12445–12452, May 2021. doi: 10.1609/aaai.v35i14.17476. URL https://ojs.aaai.org/index.php/AAAI/article/view/17476.