REPORT DOCUMENTATION PAGE

Form Approved OMB NO. 0704-0188

1. REPORT DATE (DD-MM-YYYY)	2. REPORT TYPE	3. DATES COVERED (From - To)
16-05-2022	Final Report	15-May-2021 - 14-May-2022

4. TITLE AND SUBTITLE
Final Report: Oracle Imitation for Embedded Decision Making

5a. CONTRACT NUMBER
W911NF-21-1-0243

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER
611102

6. AUTHORS

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAMES AND ADDRESSES
University of California - San Diego
Office of Contract & Grant Adm
9500 Gilman drive, MC 0934
La Jolla, CA 92093 -0934

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS (ES)
U.S. Army Research Office
P.O. Box 12211
Research Triangle Park, NC 27709-2211

10. SPONSOR/MONITOR'S ACRONYM(S)
ARO

11. SPONSOR/MONITOR'S REPORT NUMBER(S)
75035-CS.4

12. DISTRIBUTION AVAILIBILITY STATEMENT
Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES
The views, opinions and/or findings contained in this report are those of the author(s) and should not contrued as an official Department of the Army position, policy or decision, unless so designated by other documentation.

14. ABSTRACT

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	15. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Michael Yip
UU	UU	UU	UU		19b. TELEPHONE NUMBER 858-822-4778

RPPR Final Report
as of 14-Sep-2022

Agency Code: 21XD

Proposal Number: 75035CS **Agreement Number: W911NF-21-1-0243**
INVESTIGATOR(S):

Name: Michael Yip
Email: m1yip@ucsd.edu
Phone Number: 8588224778
Principal: Y

Organization: **University of California - San Diego**
Address: Office of Contract & Grant Adm, La Jolla, CA 920930934
Country: USA

DUNS Number: 804355790 EIN: 956006144
Report Date: 14-Aug-2022 Date Received: 16-May-2022
Final Report for Period Beginning 15-May-2021 and Ending 14-May-2022
Title: Oracle Imitation for Embedded Decision Making
Begin Performance Period: 15-May-2021 **End Performance Period:** 14-May-2022
Report Term: 0-Other

Submitted By: Michael Yip Email: m1yip@ucsd.edu
Phone: (858) 822-4778

Distribution Statement: 1-Approved for public release; distribution is unlimited.

STEM Degrees: 0 **STEM Participants:** 2

Major Goals: The project's major goal is to investigate a fundamentally new approach to decision making (planning) via imitation learning. The approach involves imprinting the behaviors of optimal solvers (oracles), such as Djikstra's algorithm, into neural networks to create an oracle network. The approach hypothesizes that by training a function approximator such as a neural network to mimic prior demonstrations from the oracle, we can learn embeddings of the problem and the oracles' solutions and reproduce optimal solutions across both old and new problem instances while bypassing previous speed and computational complexity barriers associated with the oracle algorithms. This idea of producing behaviors using a neural embedding was motivated by looking at how Nature encodes intelligence. The human and animal brain generates behaviors in complex dynamic environments on instinct, rather than running internal tree-search algorithms on-the-fly.

Our preliminary work has fundamentally shown oracle imitation as a computationally powerful and transformative approach to solving continuous domain planning problems. To expand this research and explore its potential to solve a wide variety of complex, computationally challenging planning problems, we will pursue the following are key research questions in this proposal:

1. What is an ideal architecture of oracle imitation to solve discrete, combinatorial problems, and how can it handle stochasticity in these systems?

2. How can oracle networks learn increasingly complex, hierarchical tasks that include nested discrete and continuous planning spaces?

3. How can oracle networks be trained to learn completely new tasks in new environments with as few queries to the oracle as possible?

Accomplishments: See report of accomplishments in the uploaded PDF report covering the entire period of the award.

Training Opportunities: Two Ph.D. students were trained during this period (1 year). The Ph.D. students were mentored by the PI weekly and in lab meetings where they were able to present their work. This professional development extended to presenting their work in university-wide research seminars attended by graduate students and faculty in the AI/Machine Learning/Robotics fields.

RPPR Final Report
as of 14-Sep-2022

Results Dissemination: Jacob J Johnson, Linjun Li, Ahmed H Qureshi, Michael C Yip, ”Motion Planning Transformers: One Model to Plan Them All”. arXiv preprint arXiv:2106.02791. (open-access via ArXiv, currently in resubmission process for another robotics conference)

The work involving discrete graph solutions is being prepared for submission in a conference venue to-be-determined.

Honors and Awards: Nothing to Report

Protocol Activity Status:

Technology Transfer: Nothing to Report

PARTICIPANTS:

Participant Type: PD/PI
Participant: Michael Yip
Person Months Worked: 1.00 **Funding Support:**
Project Contribution:
National Academy Member: N

Participant Type: Graduate Student (research assistant)
Participant: Yuheng Zhi
Person Months Worked: 6.00 **Funding Support:**
Project Contribution:
National Academy Member: N

Participant Type: Graduate Student (research assistant)
Participant: Jacob Johnson
Person Months Worked: 3.00 **Funding Support:**
Project Contribution:
National Academy Member: N

CONFERENCE PAPERS:

Publication Type: Conference Paper or Presentation **Publication Status:** 4-Under Review
Conference Name: Conference on Neural Information Processing Systems (NeurIPS)
Date Received: Conference Date: 07-Dec-2022 Date Published: 15-Dec-2021
Conference Location: virtual
Paper Title: Motion Planning Transformers: One Model to Plan Them All
Authors: Jacob J. Johnson, Linjun Li, Ahmed H. Qureshi, and Michael C. Yip
Acknowledged Federal Support: **Y**

RPPR Final Report
as of 14-Sep-2022

Partners

,

I certify that the information in the report is complete and accurate:
Signature: Michael Yip
Signature Date: 5/16/22 6:57PM

Oracle Imitation for Embedded Decision Making - Award 75035CS
Final Report

POC: Michael Yip, Ph.D.
Director, Advanced Robotics and Controls Lab
Director, Medical Robotics Collaboratory, UCSD Contextual Robotics Institute
Associate Professor, Electrical and Computer Engineering
University of California San Diego
Phone: (858) 683-3336
Email: yip@ucsd.edu

May 16, 2022

1 Overview

This final report details the work done for Award 75036CS spanning. The **overall objective** of the award is to solve challenging combinatorial optimization problems involving discrete planning on graphs, such as the traveling salesperson problem, that come up in a large variety of military and non-military applications. The **overall approach** is to describe the discrete planning problem in such a way that it can be solved using neural networks.

Previous Period: The previous period involves improving the quality of the state-of-the-art neural solver, the TSP Transformer [1] by a few modifications, including efficient training by decomposing the problem, off-policy Reinforcement Learning, a simplified network architecture, and taking advantage of the partial reward at inference time. We also pointed out a few key challenges that limit the practical applications of current neural TSP solvers, mainly related to the computational complexity and generalizability of a trained NN solver.

Current Period: Since our last report, in addition to the improvement we gain over the existing TSP Transformer, we have made substantial attempts to tackle some of the challenges we mentioned in the last period. In this last report, we detail how we scale a neural TSP solver to instances of more than 1000 nodes and improve its generalizability. We highlight the remaining challenges and promising directions to make real-world impact with the neural TSP solver.

In the following text, we will report the research and output over the **the entire period of performance**, broken into three sections: the background material, the work done in the previous periods, and the work conducted in the most recent/final period.

2 Background

2.1 Introduction to TSP

Combinatorial optimization (CO) is a topic of interest for numerous fields of study, from molecular design [2] to green logistics [3]. CO deals with finding the optimal object from a finite set of objects [4]. Some examples of CO problems are: Given a set finite number of coins of different denominations, find the minimum number of coins that can be used to form a particular value

(knapsack problem), given a collection of cities, find the network of roads that needs to be constructed such that every city is reachable and the least amount of road is constructed (minimum spanning trees). The traveling salesman problem (TSP) is also a CO problem, where the objective is to find the shortest path to visit all cities and return to the starting city given a set of cities. The cities and allowed paths between pairs of cities are represented as a graph $G(V, E)$, where V is the set of nodes and E is the set of edges. We first focus on this basic form of TSP, while modification to this problem, e.g., adding or relaxing constraints, can be considered in future work to suit the need of specific practical applications.

In this summary, we first briefly introduce typical classical solvers of the TSP problem and modern solvers using machine learning. Next, we present our method to solve an easier but similar CO problem, the Motion Planning Transformers. We also provide the first approach we will be testing to solve the TSP problem. Finally, we briefly discuss techniques that can be used to further boost performance beyond this first approach.

2.2 Existing Solvers of TSP

Despite its computational complexity, TSP in its original form has been a relatively well-studied problem. Before the emergence of deep learning, many exact methods or heuristic-based methods have been proposed to solve it at different levels of computational costs. We will cover the most typical ones among them. These methods tend to solve different TSP problems in isolation. Recently, due to the development in deep learning, especially Graph Neural Networks (GNNs), some works have been focusing on using neural networks to solve TSPs generated from a similar distribution to the training data. These methods have been shown to provide significant acceleration compared to classical methods but still lag behind in terms of quality of the solution and generalizability to problems outside the distribution of the training data.

It is also worth noting that the evaluation benchmark of TSP solvers is mostly on highly abstracted toy problems, which classical solvers are extensively developed on. For example, many algorithms are only evaluated on graphs whose nodes are points on a 2D Euclidean space, an edge existing between each pair of nodes, and edge costs being the Euclidean distances between the end points. It remains unclear whether classical solvers can maintain their advantages when the problem instances become more challenging, e.g., when the graph structures become extremely irregular and constraints become more strict. So developing a more thorough benchmark that aims at solving specific real-world problems is also a potentially valuable research direction.

2.3 Classical Solvers

Dynamic Programming can be used as exact solvers to a series of sequencing problems [5], which includes TSP. While it achieves provably optimal solutions, DP methods have exponential time complexities $O(n^2 2^n)$, which becomes impractical for $n > 40$.

Gurobi is a general-purpose commercial software for Integer Programming [6], which can be adapted to solve TSP exactly. It provides efficient implementations of standard techniques for solving integer programming problems such as Cutting Planes and Branch-and-Bound.

Concorde is a specialized TSP solver based on Integer Programming [7]. It is considered the fastest exact TSP solver for large problem instances so far.

2-opt is a heuristic algorithm to solve TSP [8]. 2-opt iteratively attempts to lower the total cost by swapping any pairs of nodes on the solution. This method has a strong inductive bias from Euclidean space because it tries to lower the cost by rerouting two segments that cross each other, so it is not surprising that it can also be extended to vehicle routing problems which are roughly in a 2D Euclidean space. 3-opt and 4-opt [9] are generalized versions of 2-opt that iterates over triplets of nodes.

2.4 Machine Learning-based Solvers

The TSP is an NP-hard problem, but this is considering the worst-case scenario. In many situations, problem-specific structures [10] or relaxations to conditions [11] have been used to propose a solution. For many real-world tasks, we can leverage the data from past solutions to propose a solution for a future scenario. Consider a CO problem where a delivery company has to arrange a given number of boxes on a truck efficiently. Each day the number and the shape of boxes may differ, and traditional algorithms might be used to solve this problem. Learning methods can be used to infer the patterns within these datasets and later solve a future stacking problem. Such an approach was recently used to predict protein structures [12] by learning interactions between sub-structures from previous models. Traditional learning models such as a multi-layer perceptron (MLP) operate on vectors and cannot be applied to inputs of varying sizes, which is often the case for TSP problems. Network architectures such as recurrent networks, long short-term memory [13] can handle inputs of different sizes. Thus, one of the first attempts to solve TSP using deep learning, Pointer Networks, employs LSTMs to model TSP as a sequence-to-sequence problem [14]. However, recurrent networks require recursions during inference which can slow the learning process. Also, since the TSP problem is defined on a *set* of nodes and edges, its solution should be invariant to the *order* of the input nodes and edges. MLPs and LSTMs are sensitive to the order of their inputs and, therefore, are not ideal options for TSP. Even though *permutation-invariance* can be approximated by applying data augmentation during training [15], it is often preferred as a property of the model by design.

Graph Neural Networks A recent form of network architecture called Graph Neural Networks exploits the underlying graphical structure of the data to make inferences. Unlike MLP's, GNN's operate on sets and can accommodate inputs of varying sizes, and its output is permutation-invariant to its input, which makes it a natural choice for TSP or even CO problems on graphs in general. Kool et al. [16] tackled TSP among other routing problems using Graph Attention Networks [17]. S2V-DQN formulates the TSP problem as a node prediction task using a partial output sequence and uses a GNN to solve it [18]. GNNs can also be trained to predict the probability of an edge being in the solution [19, 20].

Although developed independently from GNNs, Transformer [21] can be considered as a form of GNN where all vertices are connected to each other [22]. Transformer networks, i.e., neural networks that employ the self-attention operation, have become one of the state-of-the-art models used in natural language processing and are also being used for semantic segmentation of images [23, 24]. It has been recently found that Transformers can serve as encoders of the TSP problem [25] and also as direct end-to-end solvers [1, 16].

Reinforcement Learning Early attempts to TSP train the neural networks using supervised learning. Solutions given by classical solvers are used as correct labels that are imitated by the NN. However, this approach is not feasible for larger problem instances because solving them may require exponentially exploding computational time depending on the input size. Many works have

explored the possibility of using reinforcement learning to train a neural TSP solver without labels. The authors of [15, 26] trained the Pointer Networks [14] using reinforcement learning; the total path length were used as the reward. S2V-DQN [18] trains a deep Q network to estimate the state-action value function, where the state is a partial solution and the action space is the remaining nodes. It introduces intermediate rewards on top of the total path length, such as furthest node insertion, to speed up the training procedure.

Test-time Search Instead of using the output of a neural network directly as the solution, some methods combine deep learning with searching algorithms to boost test-time performance. In this case, the neural network may output some intermediate results that help in searching. The most common searching algorithm beyond naive greedy sampling may be beam search, which is widely used in the inference time of natural language processing models. These methods often train probabilistic models that output step-wise distributions and rely on beam search to optimize the overall likelihood of sequence [1, 19, 27]. It is also possible to embed deep learning in heuristic searching algorithms. The NN in [28] learns a heuristic to help the 2-opt operation, which is trained using reinforcement learning. [25] implements a similar idea using Transformer networks. Inspired by AlphaGo [29], Abe et al. and Xing et al. [30, 31] first employ Monte-Carlo Tree Search at test-time to search for a solution using heuristic value predicted by NNs. Although conceptually promising, these MCTS methods have yet fully transferred the success of MCTS in Go game to TSP.

A major takeaway is that Transformers are becoming increasingly popular in solving graph CO problems, especially TSP. This is possibly due to the bag of tricks extensively developed to stabilize the training of Transformers on natural language processing tasks that could also generalize to graph CO problems. So we will choose Transformer as the network architecture we start within this project. Rather than jump to solving the TSP problem, we first investigated if Transformers can solve an easier CO problem of finding the shortest path for a given graph in the next section. In this problem, we can still rely on classical solvers to provide near-optimal solutions as ground-truth to supervise model training.

3 Description of Work in the Prior Periods (May 15, 2021 to Nov 14, 2022)

3.1 Motion Planning Transformers

Path planning for mobile systems on a 2D plane can be formulated as a CO problem. For a 2D navigation system, the planner is given a 2D costmap represented by an $N \times N$ vector, a start location, and a goal location. The objective is to propose the shortest collision-free path from the start to the goal location. Since there are a discrete number of grids, a countable number of paths exist from the start to the goal location, and the objective is to choose the optimal path. Some of the traditional planners used are A*, and later papers added heuristics to reduce the planning time. [32–34]. But these methods require extensive hand tuned heuristic functions for planning. Recent papers have proposed neural network-based models using MLP architecture [35] to solve this problem but are not scalable to maps of varying sizes. We propose an algorithm using Transformers for this problem called Motion Planning Transformers (MPT). An outline of our method is given in Fig. 1.

Path planning requires an understanding of both the global and local layout of obstacles in the planning space. In MPT, the initial convolutional layers embed patch information (local structures)

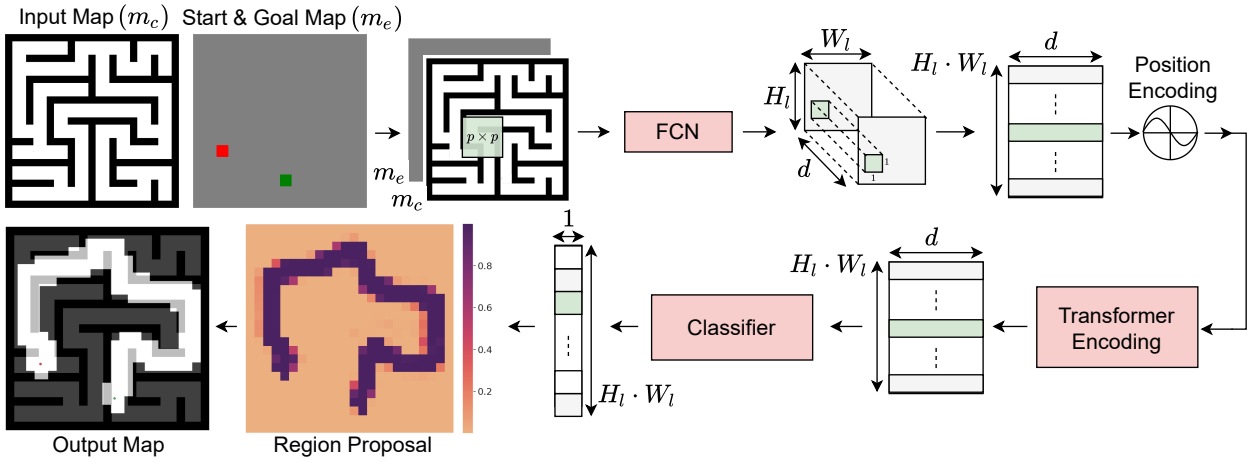


Figure 1: Overview of MPT Module. (Start from the top left and move clockwise) The input map and the start and goal encoded map are concatenated together and passed through a Fully Connected Network (FCN). The output of which is reshaped and passed through the standard Transformer module. The classifier predicts the probability of each of the tokens from the encoder output to create the mask superimposed on the output map. The light green patches throughout the pipeline represent how a patch $p \times p$ remains connected with the output of the classifier.

Table 1: Comparing planning accuracy, median planning time, and median number of vertices in the planning tree for the Point Robot Model on environments of the same size as the training data.

Environment	Random Forest			Maze		
	Accuracy	Time (sec)	Vertices	Accuracy	Time (sec)	Vertices
RRT*	100%	5.44	3227.5	100%	5.36	2042
IRRT*	100%	0.42	267	100%	3.13	1393.5
MPT-RRT*	97.79%	0.20	251	98.96%	0.83	615
MPT-IRRT*	97.79%	0.07	133	98.96%	0.74	557
MPT-RRT*-EE	100%	0.22	266	100%	0.84	595.5

into a latent space, while the Transformer Module aggregates information from all these embeddings to propose a valid region. The ability of Transformers to combine all the local embeddings to infer the global layout makes it an ideal architecture for motion planning applications. Simple path planning algorithms such as A*, hybrid A* [34], RRT* can now be applied to this selected region to find the valid path. In our work, we used the RRT* and Informed RRT* [36] planner to generate the final path.

To test the planning capabilities of our method, we evaluated the model on randomly generated maps from two different kinds of synthetic environments. The first environment is called the Random Forest, where 100 circular and square objects are randomly placed in different positions and orientations on the map. The second environment is called the Maze environment. A perfect maze is generated using the randomized depth-first search. We call the planners MPT-X, where X is the traditional planner used to plan the path. We also implemented a planner that switches between exploiting the masked region produced by MPT and explores the region unmasked region, which we call MPT-RRT*-EE. The results are summarized in Table 1.

These results show that Transformers have the ability to solve CO problems.

3.2 Transformer models and towards solving the combinatorial optimization problem

The previous work involves scoping the planning problem at hand (we use Traveling Salesperson, or TSP, as the target problem), determining a problem formulation suitable for solving the problem, performing an extensive literature search across TSP and neural planning approaches, and identifying novel neural architectures that lend itself towards solving the TSP problem. We found that a bridge to solving the combinatorial problem was to solve the discrete planning problem using neural planning, something that had not yet been deeply explored. We considered the use of Transformers, which were popularized and showed remarkable acuity in the neural language processing field for picture captioning and sentence completion, as potential high-level architecture. We first showed that Transformers can be used to very effectively solve shortest path problems within a graph, and set up a strategy to begin solving the discrete planning problem that involves the combinatorial aspect of visiting every node.

The TSP Transformer proposed in [1], closely following the design in [16], is composed of two parts: an encoder network that parses the graph and transforms it into latent state vectors and a decoder network that outputs a solution in the form of a permutation of node indices. The encoder network is a feed-forward Transformer network that does several consecutive layers of multi-head self-attention operations. The decoder network is a recurrent network that takes the output of the encoder and repeatedly chooses one node from the remaining/unvisited nodes at each timestep. A key design that allows the whole network to be trained end-to-end is that the recurrent decoder does *not* output a node index at each timestep; instead, it outputs the *likelihood* of each remaining node to be the optimal next node and sample one node according to the distribution defined by the likelihoods. Then the log-likelihood of the whole solution sequence is designed to be the sum of the log-likelihood of all steps. With the predicted log-likelihood of the solution, and the total length of the trip as the reward signal, the network is trained using the REINFORCE algorithm, which is based on the Policy Gradient Theorem.

3.3 Efficient Training by Decomposing the Problem

One issue we observed with the TSP Transformer is that it converges very slowly. On TSP instances consisting 50 nodes, it takes roughly 1.5 days to reach an optimal gap of 1%, but then it takes another 19 days to achieve the claimed performance in the paper. This could attribute to the fact that the REINFORCE algorithm is well known for having large variance in the gradients it estimates. However, this should have been largely resolved by subtracting a baseline reward from the actual reward. We believe this issue may come from a potentially sub-optimal way of implementing reinforcement learning (RL) for TSP: even though the decoder network has a notion of *timesteps*, the timestep in the formulated RL problem is only *1*. Specifically, the whole solution consisting of many steps, despite being output in a recurrent manner, receives only one reward at the very end. This results in an *credit assignment* problem, which further caused inefficient training.

To this end, we propose to modify the way of formulating an RL problem for TSP. Instead of formulating the whole solution as one action in the RL problem, we decompose the RL problem into individual timesteps, whose action space is no longer the (exponentially large) space of all permutations of the nodes, but just the space of all remaining nodes. By making this modification, we are able to provide each step with an immediate reward, which is the distance between the current node and the next node. This allows us to calculate the policy gradients with the cumulative returns of all actions (selection of the next node), which is more informative than the overall length.

More importantly, this modification allowed us to apply off-policy RL algorithms and simplify the network architecture for faster training.

3.4 Off-policy RL and a Simplified Network Architecture

Now we are considering the decomposed TSP problem: given the graph of cities and a partial tour, what is the next optimal city to visit? Since this problem has more than 1 timestep, it becomes more convenient to apply an Off-policy RL algorithm to it than the original formulation that wraps the whole sequence as 1 step. The benefit of using an Off-policy RL algorithm is that it is always converging to the optimal value function no matter what the current executed policy is, meaning it may converge faster than an on-policy algorithm like REINFORCE. We chose the most basic off-policy deep RL algorithm, the DQN algorithm, because we observed that a large number of the modern *best* deep RL algorithms have only been evaluated on image-based/state-based control tasks, which means their superiority over DQN may not necessarily transfer to the combinatorial optimization domain.

The DQN algorithm uses a different type of model output. Instead of outputting the likelihood of each remaining city being optimal, it predicts the return of choosing each remaining city, i.e., the state-action value function. Also, it does not rely on fresh rollouts of the policy to train the network, but takes samples from a replay buffer of past experiences, making it stabler to train.

Despite the fact that DQN being an off-policy algorithm that may converges faster, the usage of DQN removes the need for a decoder network that outputs action likelihood. Therefore, we are able to only keep the encoder of the TSP transformer and getting rid of the *recurrent* decoder, which can itself be hard to train. We make two modifications to the encoder to make it fit for the DQN algorithm:

- adding a mark to each city indicating whether it has been visited, or if it is the start or the end of the partial tour, and
- adding a shared linear layer to predict the return of choosing each node as the next city.

3.5 Taking Advantage of the Partial Reward at Inference Time

One often overlooked fact of neural solvers for TSP is that its inference strategy does matter. Unlike most tasks solved by neural networks, a TSP network does not greedily output the most likely next node or the next node with the highest predicted return; it uses the predictions as a kind of heuristics to guide its search strategy. Specifically, it keeps a large *set* of the partial tour that look best so far, try predicting the next action from each of them, and again collects a new set of the best candidate partial tours from all descendants of the previous candidate tours. When a network predicts the likelihood of each node being optimal, like the original TSP Transformer does, its search strategy works in the learned likelihood space, which may not even be accurate. Its final solution is merely a solution that has a high total *learned* likelihood, instead of high true reward. The proposed DQN-based network predicts the expected return of each node, which is in the same space as the true reward. Therefore, the search strategy of the new network can operate in the space of the true reward, i.e., at each timestep, we maintain the candidate tours that have the highest sum of partial tour length and expected return in the future. Note that a component of the heuristic, *the partial tour length*, is no longer a learned metric but actual reward. This leads to a more accurate and practical inference time search strategy.

4 Preliminary Results

We will show the effectiveness of these modifications by some preliminary results. We trained the proposed Transformer network using DQN on TSP instances of 50 cities (TSP50). The evaluation metric is the average percentage gap (the smaller the better) between the NN solutions and the solutions given by Concorde, the best solver of TSP so far. During this, we discovered some minor bugs in the benchmark code of the original TSP Transformer and corrected them, which made Concorde's results slightly better. We run beam search of the same beam sizes on the proposed and the original TSP Transformer. Using 1.5 days of training time and 2500 beam size, the DQN-based Transformer reached an optimality gap of 0.0005% on TSP50, while the original TSP Transformer reached an optimality gap of 0.017% after 20.3 days of training. Figure 2 and 3 present the best and worst solutions given by the proposed Transformer.

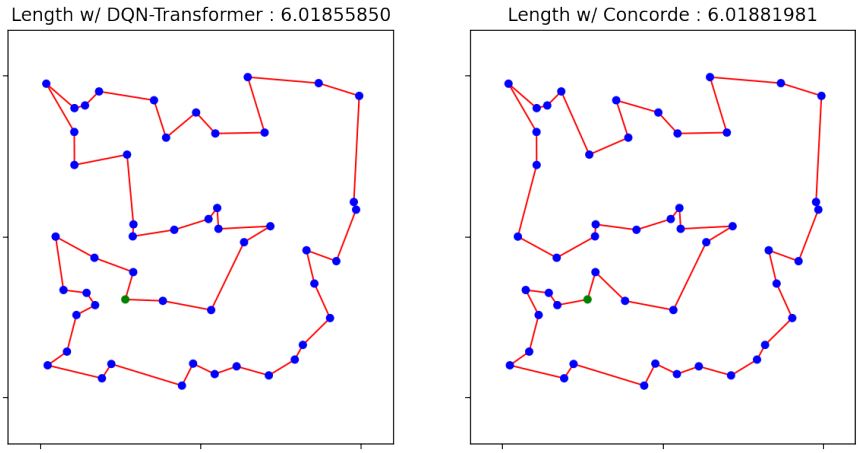


Figure 2: The TSP50 instance to which the DQN-based Transformer produced the *best* solution in terms of the solution's percentage performance gap compared with Concorde. On this instance, the DQN-based Transformer beat Concorde.

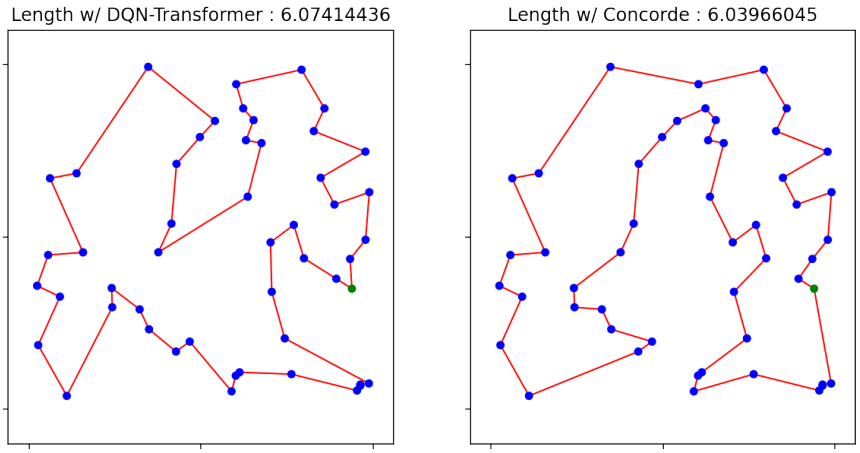
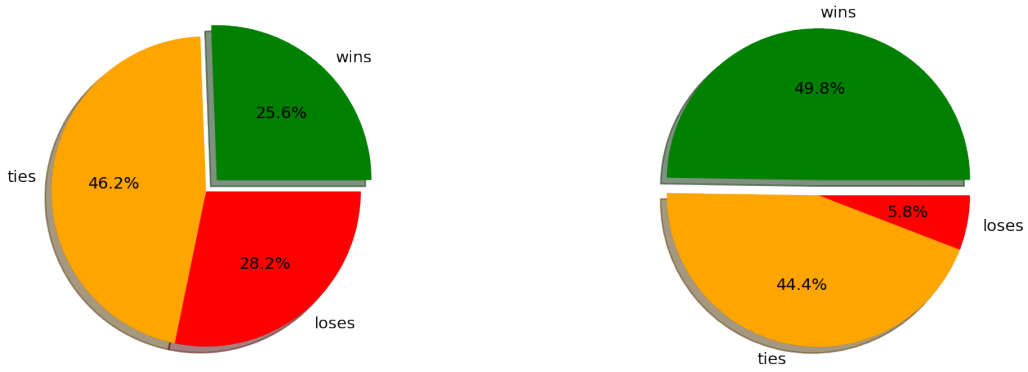


Figure 3: The TSP50 instance to which the DQN-based Transformer produced the *worst* solution in terms of the solution's percentage performance gap compared with Concorde. On this instance, the DQN-based Transformer is beaten by Concorde.

Another way of evaluating the performance of the neural solvers is to see among all the problem instances, in how many they beat the classical solver Concorde. In the 10,000 TSP instances we tested, the proposed Transformer wins 4975 times, loses 584 times, ties 4441 times; the original Transformer wins 2559 times, loses 2824 times, ties 4617 times. The proposed DQN-based Transformer wins Concorde more often than the original one by a clear margin.

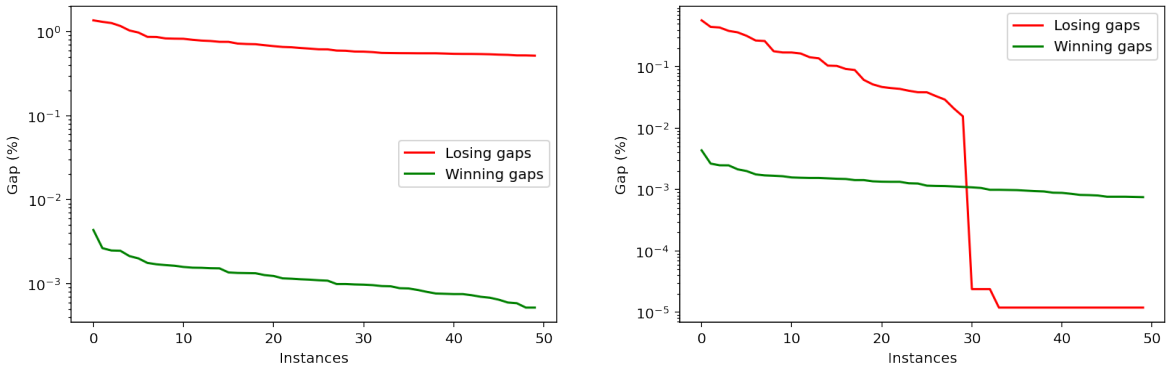


(a) Original TSP Transformer

(b) DQN-based TSP Transformer

Figure 4: The percentage of wins, ties, and loses of the original and the proposed TSP Transformer when they are compared against the solutions given by Concorde.

A seemingly unusual result is that the DQN-based TSP Transformer has a positive performance gap (0.0005%) to Concorde even though it wins and draws with Concorde much more often than losing. Figure 5b takes a closer look at the reason behind this. Essentially, when the DQN-based TSP Transformer wins Concorde, it does only by small margins; however, when it loses, it loses by fairly larger margins, which offset the winning margins when averaged. This shows the robustness of Concorde, in the sense that its sub-optimality solely comes from the fact it converts floating-point numbers to fixed-point before computing an optimal solution via integer programming. Concorde leaves little room for improvement on small instances for neural solvers, highlighting the most advantageous domain for neural solvers may be much larger instances or those with more complicated constraints.



(a) Original TSP Transformer

(b) DQN-based TSP Transformer

Figure 5: The top-50 winning and losing gaps (absolute value) of the original TSP Transformer and the proposed one, when compared to Concorde.

5 Description of Work in the Final Period (Nov 15, 2021 to May 15, 2022)

5.1 Scaling the Neural TSP Solver to Thousands of Nodes

Even though we made the *training* of the TSP Transformer more efficient by decomposing the problem and thus being able to use a more compact network architecture, the *inference* complexity of the upgraded network remains $O(n^2)$, both in terms of time and space. This becomes the biggest bottleneck to apply the NN solver to real-world problems, which constantly involve thousands or tens of thousands of nodes. Partially inspired by research in natural language processing with Transformers, we attempt to lower the computational footprint of the neural solver by compressing the pooling operations in the self-attention mechanism.

5.1.1 Background: the Complexity of the Multi-head Self-attention Mechanism

To make it easier to see why the multi-head self-attention mechanism employed by the Transformer networks is computationally expensive and how different methods are proposed to improve it, we briefly introduce its mathematical fundamentals.

The multi-head self-attention mechanism is used in each Transformer layer. Assuming the input embedding of the i-th layer is $H_{i-1} \in \mathbb{R}^{n \times d_h}$, where n is the length of the sequence (e.g., the number of nodes in our context), the output of the operation is

$$\text{MultiHead}(H_{i-1}) = \text{Concat}(\text{head}_1, \text{head}_2, ..., \text{head}_M) W_i^O, \tag{1}$$

$$\text{with head}_m = P H_{i-1} W_{i,m}^V = \text{softmax}\left[\frac{H_{i-1} W_{i,m}^Q (H_{i-1} W_{i,m}^K)^\top}{\sqrt{d_q}}\right] H_{i-1} W_{i,m}^V, \tag{2}$$

where $W_i^O \in \mathbb{R}^{M d_v \times d_h}$, $W_{i,m}^Q, W_{i,m}^K \in \mathbb{R}^{d_h \times d_q}$, $W_{i,m}^V \in \mathbb{R}^{d_h \times d_v}$ are learned matrices and d_q, d_v, and d_h are hyperparameters that decide the number of parameters in the model. Q, K, V stand for *query*, *key*, and *value*, respectively. Concat$(\cdot)$ stacks M matrices $\in \mathbb{R}^{n \times d_v}$ into one matrix $\in \mathbb{R}^{n \times M d_v}$; softmax$(\cdot)$ applies the *softmax* operation along the second axis of the input matrix $\in \mathbb{R}^{n \times n}$. Each row of the output matrix of the softmax operation, a.k.a. the attention matrix, is a probability distribution that sums up to 1. In the original version of the multi-head self-attention mechanism [16], the input embedding H may be composed of three parts, Q, K, V; the encoder network is designed to take identical Q, K, V but the decoder network takes different Q, K, V. In this project, however, since we do not need a decoder network, we simplified the notation by replacing Q, K, V by H. More simplification is done by using $d_q = d_v = d_h$.

The most computation comes from the calculation of the attention matrix P, which involves the multiplication of an $n \times d_q$ matrix and an $d_q \times n$ matrix, making its time and space complexity both $O(n^2 d_q)$. Even though being parallelizable on off-the-shelf parallel computing devices, e.g., GPUs, the quadratic relationship with n is a major computation bottleneck for Transformers, as it quickly becomes unacceptable for both computation time and memory usage as n gets close to the order of a thousand, which is only a moderate size for real-world problems. Another fact that limits the performance of our current method is that the whole network needs to be queried for $n - 1$ times, recurrently, to solve a problem instance of n nodes at test time.

5.1.2 Efficient Self-attention Mechanism via Top-K Pooling

We attempt to significantly lower the computational cost of the self-attention mechanism by reducing the number of entries in the attention matrix P with Top-K pooling. The idea is to reduce the

number of *keys* (i.e., $(H_{i-1}W^K_{i,m}) \in \mathbb{R}^{n\times d_q}$). A previous work attempted to learn a linear mapping $L \in \mathbb{R}^{n\times K}$ that projects the n keys into K keys by $(LH_{i-1}W^K_{i,m}) \in \mathbb{R}^{K\times d_q}$ [37]. However, the learned mapping can only be applied to problems with a fixed n, the same n used during training. This property is undesirable in our context as the number of nodes in TSP problems is highly variant.

To this end, we adapt a more flexible pooling strategy previously used in point cloud [38] and graph analysis [39, 40], *Top-K pooling*, into the computation of the attention matrix. We further simplified Top-K pooling by removing the gate operation so that it does not involve any extra parameters. After this modification, the feature of each head is computed by

$$\text{head}_m = \text{softmax}\left[\frac{H_{i-1}W^Q_{i,m}\left(\text{TopK}\left(H_{i-1}\right)W^K_{i,m}\right)^\top}{\sqrt{d_q}}\right]\text{TopK}\left(H_{i-1}\right)W^V_{i,m}, \tag{3}$$

where $\text{TopK}(H_{i-1}) \in \mathbb{R}^{K\times d_h}$ selects the largest K values out of n values in each channel of H_{i-1} and sorts them in a descending order. Now the major computation of the attention matrix becomes multiplying an $n \times d_q$ matrix and a $d_q \times K$ matrix, which costs $O(nKd_q)$ of time and space. By using a fixed $K \ll n$, we achieve linear time and space complexity w.r.t. n.

The Top-K pooling strategy increased the maximum number of nodes an NVIDIA 3090 graphics card can train from 700 to 4000, with a batchsize of 64 and $K = 16$. The training speed for TSP problems with 500 nodes is 2.81× of that of the original method. The optimality gap of the solutions remains almost unchanged.

5.1.3 Faster Inference With Lightweight Recurrent Head

Our previous network was designed to take the coordinates $x_c \in \mathbb{R}^{n\times 2}$ and the identity labels $x_s \in \mathbb{Z}_2^{n\times 5}$ of the nodes together as input. Every row of x_s is a one-hot vector, indicating which one of the following identity groups the node belongs to: the *start* node of the partial tour, the *last* node of the partial tour, a *visited* node other than the start or the last, an *unvisited* node, or a *padding* node. At every timestep, i.e., after deciding the next node to visit, x_s changes according to which node is visited while x_c remains unchanged. However, even though only part of the input is changed, it needs to be re-propagated through the whole Transformer network to get the next decision. This becomes a major bottleneck at inference time as it means the whole neural network needs to be queried for $n - 1$ times recurrently when solving a TSP instance of n nodes.

To this end, in addition to the Top-K pooling strategy, we propose to partially separate the Transformer network into two parts: a coordinate encoder that takes only x_c as the input and generate an embedding of the problem, and a joint encoder that takes the embedding and the identity labels of the nodes in the current partial tour, and compute the Q value of each node. The coordinate encoder needs to be queried only once for every TSP instance as x_s does not change, while the joint encoder is designed to be lightweight so that it can response to recurrent queries efficiently.

By designing the top 2 layers of the Transformer network to be the lightweight recurrent head, the inference time to solve a TSP instance with 200 nodes decreased from 63.07 seconds to 22.12 seconds, using a searching beam size of 2500. The optimality gap of the best model during training increased from 2.147% to 2.977%. One possible reason that could have led to the slight performance drop is that, with the modification, the network does not *directly* learn a joint embedding of x_c and x_s. A straightforward fix would be to duplicate x_c and feed it to the recurrent head, in addition to x_s, even though this means adding negligible overhead to the first layer of the head.

In combination with the Top-K pooling strategy, we are observing 8× overall acceleration compared to our previous design and more than 5× less memory usage, while keeping a comparable quality of solutions.

5.1.4 Generalizing to Different Numbers of Nodes

Besides performance in terms of speed and memory usage, another valued property of a neural TSP model is the generalizability. One popular scheme to test the generalizability of a model in previous works is to train it on problems of a certain number of nodes before evaluating it on problems of different numbers of nodes. A typical setting is to train a model with 50 nodes and test it on 20, 50, 100, sometimes 200 nodes. An overlooked issue with this scheme is, there is not any design in the training method or the network architecture to help the network trained on a particular number of nodes generalize to other numbers. Note that neural networks are expected to only generalize to test data that has a similar distribution to the training data, which brought its success in image and natural language data, where test and training data are extracted from similar distributions. However, when it comes to TSP problems, even when we only consider the problem instances generated by uniformly sampling points on a 2D unit square, different numbers of nodes can result in drastically different densities and local geometrical properties, making it challenging to transfer the experience learned with a specific number of nodes to a very different scale.

Thanks to the fact that we designed the network to estimate the optimal tour length starting from a certain node, as opposed to the probability of a node being the optimal step in most existing works, we are able to expose the generalizability issue of a neural TSP model in a straightforward way. We considered the problem how the optimal cost, $C(V_n = \{v_1, v_2, ..., v_n\})$, changes in expectation as the number of nodes n increases. We derived an estimate of the order of the expected optimal cost increases as n increases, for sufficiently large n, and tested whether our model would scale in a similar manner.

Proposition 1. *The expected optimal cost of an Euclidean 2D TSP instance of n nodes, $C(V_n)$, is $O(\sqrt{n})$ for any sufficiently large n.*

In addition to the sketch proof we provide in the appendix, this proposition aligns with results in previous works [41–43]. Following these works, [44] achieved a famous result indicating the existence of a β-constant, i.e., the expected optimal cost divided by $\sqrt{n}$ as n approaches infinity will converge to a constant value β.

These results pose a requirement to any neural TSP model, that it should roughly scale its (implicit or explicit) estimation of the total tour length by $\sqrt{n}$ seeing different numbers of nodes. It is convenient to evaluate our neural TSP model based on this standard since our neural network explicitly estimates the optimal tour length. We feed a trained neural TSP Transformer with random TSP instances of different numbers of nodes and plot out its estimated optimal cost in Figure 6. We also plot the function $\sqrt{n}/2$ as a comparison. As the figure depicts, the network has a good variation in the range $1-50$, which is what it is trained on. As soon as the number of nodes goes out of its training domain, it stops scaling its estimation on an order of $\sqrt{n}$ and yields almost a constant estimation for problems larger than 100 nodes. This means it is hard for a neural TSP model to generalize to problem sizes it has never seen (at least with the existing network designs which do not explicitly take generalizability into account).

Besides potential issues in network design, we believe another major reason the network fails to generalize is it has never been trained on problem instances of more nodes. This problem has some historical origins in earlier works. To the best of our knowledge, maybe due to the ease of software design, no previous work trained a neural network with TSP instances of different sizes, despite

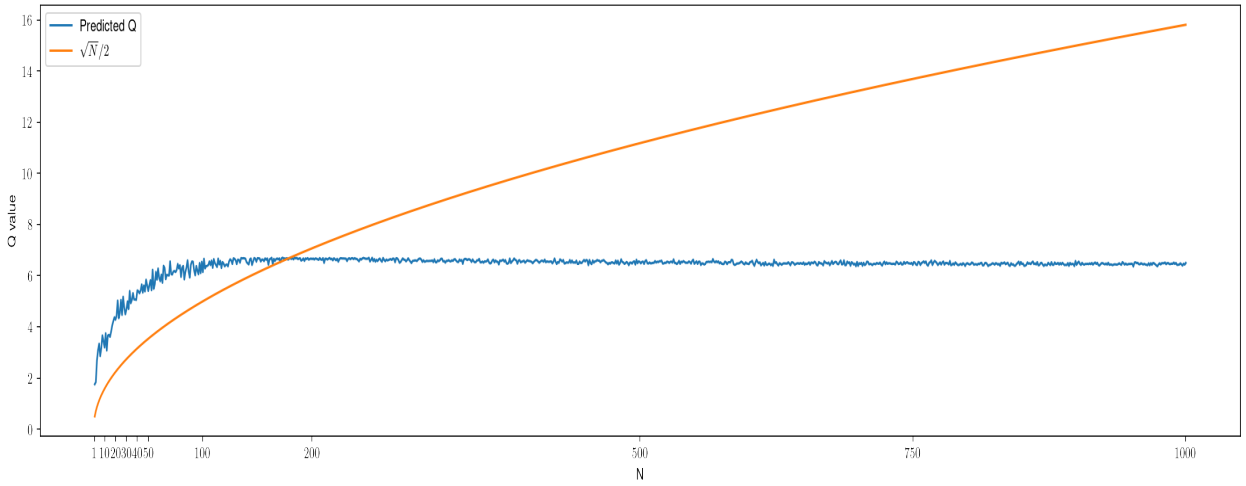


Figure 6: Estimated optimal cost by a TSP Transformer trained only on TSP instances of 50 nodes.

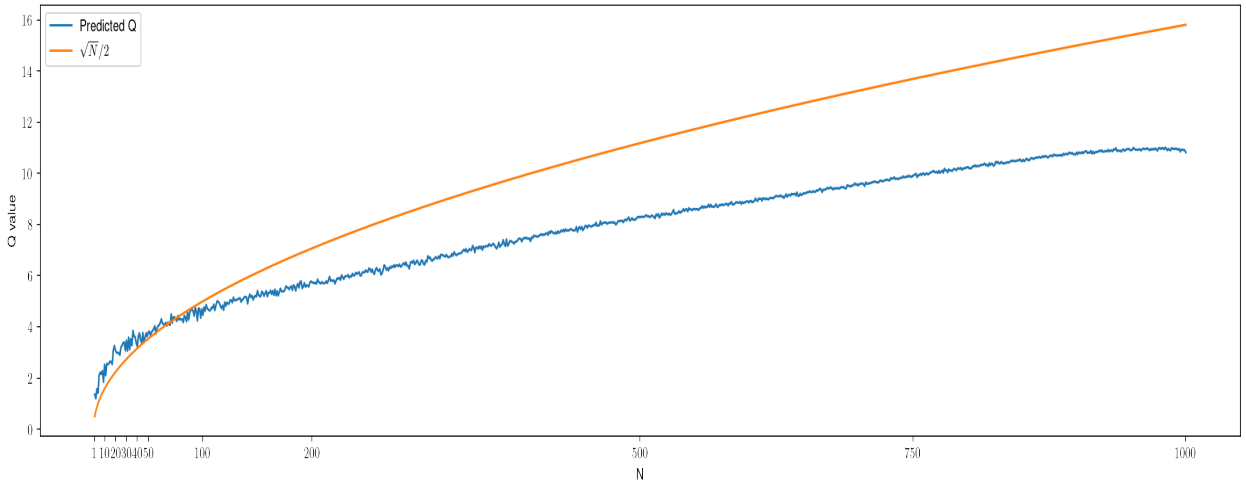


Figure 7: Estimated optimal cost by a TSP Transformer trained on TSP instances of 3 to 500 nodes.

it being a rather common practice to train language models with sentences of different lengths in natural language processing research with the help of *padding* tokens. To this end, we introduced padding nodes into our method, enabling our network to be trained with TSP instances of different sizes in the same batch. We trained a TSP Transformer with instances of random sizes uniformly drawn from 3 to 500. We also adapted our beam search method accordingly to enable *inferencing* multiple TSP instances of different sizes concurrently.

After these modifications, we evaluated the network's generalizability using the same method above (Figure 7). It yields a better trend in scaling its estimation by $\sqrt{n}$, even for problem sizes outside its training distribution (501-1000), indicating training with variable number of nodes is a promising method to partially resolve the generalizability issue of neural TSP models.

5.1.5 An Attempt in Improving the Validation Method

Unlike supervised training tasks, for TSP, it can be non-trivial to efficiently validate a model and compare it against a best model before. A typical method supervised training tasks use is to

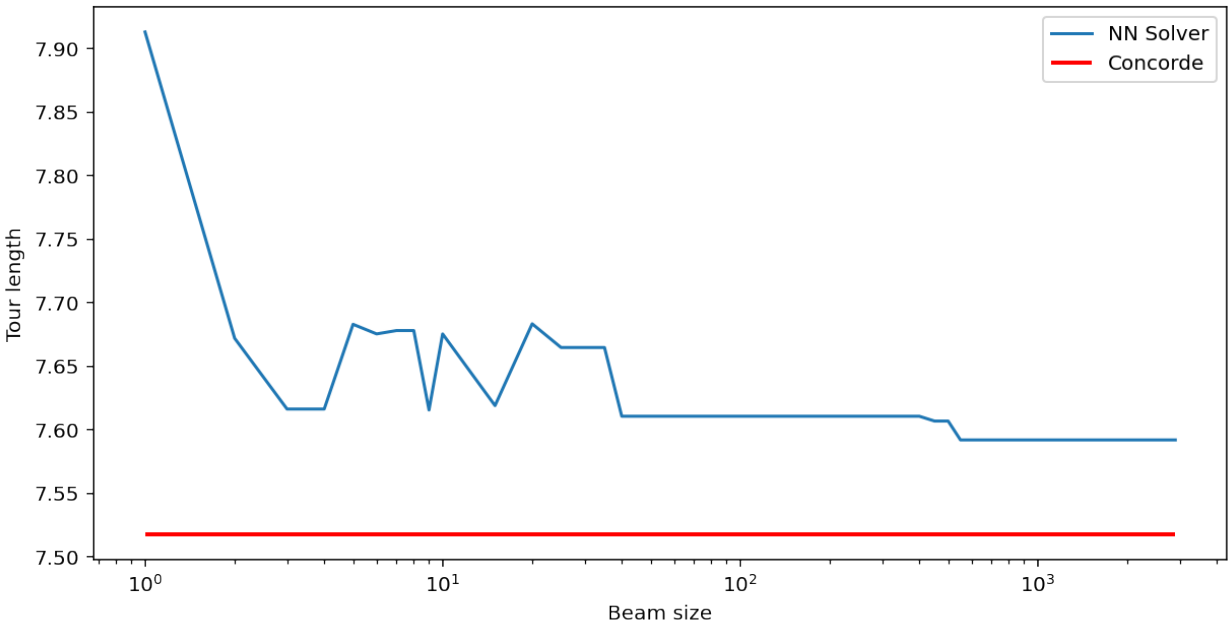


Figure 8: The relationship of the beam size and the solution quality of a TSP100 instance, model validated by counting winning instances.

separate a validation set from the training set, and use the model's performance on the validation set to determine if it is better or worse than other models. However, a couple of reasons make this scheme less ideal for TSP than it sounds. Firstly, TSP problems are in fact very diverse, in a sense that a minor change in the location of one point could result in a drastic change in the solution. This makes it hard to say whether a better model for one or some instances is really better than another model for other instances. Secondly, solving TSP problems using NNs heavily depend on beamsearch with a large beam size (more than 1000), which makes it extremely inefficient to test a model on a large validation set; meanwhile, if one chose a small beam size for validation, the performance may not reflect the true ability of the model. Finally, it is hard to say whether a model that output best solutions most often but occasionally gives a large optimality gap is better than a model that consistently output near-optimal solutions.

We made an attempt in changing the metric of validation. Instead of considering the model the generates that least average optimality gap as the best, we let the current model *compete* with a previous model on a newly generated set of TSP problems. We keep the one that wins more problems. A new model needs to win by a pre-set margin to replace the previous model. One thing we observed when using this validation is that the current best model gets replaced much more often than before. It turned out this selection method left us with models that are less *consistent*. We test the consistency of a model by using it to solve the same TSP instance with increasingly large beam sizes and plot the relationship between the solution costs and the beam sizes (Figure 8). As a comparison, a model validated in the previous way by optimality gap is shown in Figure 9. The model selected by counting winning instances does not always perform better when the beam size is increased, indicating less consistent estimation. This may be due to the fact that in the new validation rule, it is okay to be very wrong in some cases, as long as the model is better than another one on most cases.

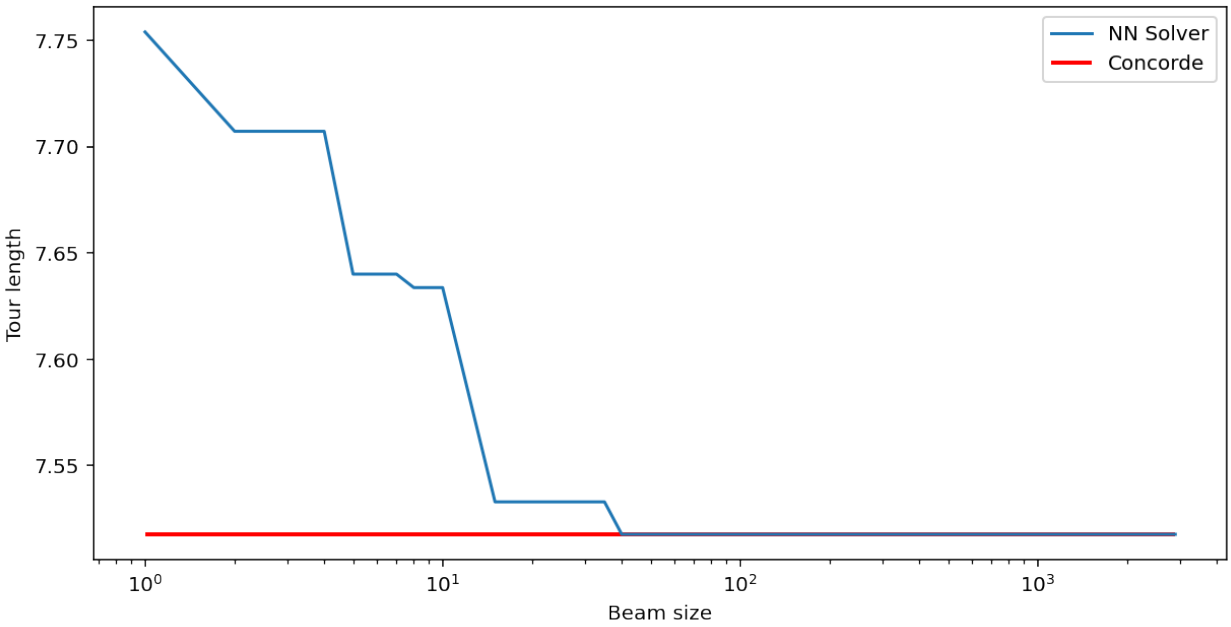


Figure 9: The relationship of the beam size and the solution quality of a TSP100 instance, model validated by average optimality gap.

6 Discussion

Following the work done in this project, we believe there are a few directions that can lead to promising improvement in the field or that are worth setting up standards in the community.

Validation Metric is still an open problem due to the large computational expense of existing neural TSP methods and the variety of TSP problems. More consistent and efficient validation methods and metrics need to be discovered to select more reliable neural TSP models during training. Various validation schemes may be explored according to specific application-wise needs.

Inference-time Search Algorithm is a module in the pipeline that has not been fully explored. Most existing works assume direct inference or beam search. More sophisticated methods with a strong mathematical foundation may bring fundamental improvement in the field. Some existing works are starting to consider Monte-Carlo Tree Search as an alternative; however, how to efficiently implement these search algorithms that involve more control flow than beam search on parallel computing devices is yet to be solved.

Working with Different Constraints is a desired ability that will bring tremendous application value to neural TSP models. It means a model trained on one variant of TSPs can be readily adapted to another variant with slightly different constraints, or even better, solve general combinatorial optimization problems other than TSPs. This may require the network not to directly learn the task itself, but learn to predict some intermediate quantities that are more generally useful among different variants of TSP. Few-shot learning techniques may also be utilized to adapt a model to a different task.

References

[1] X. Bresson and T. Laurent, "The transformer network for the traveling salesman problem," *arXiv preprint arXiv:2103.03012*, 2021.

[2] J. P. Knight and G. J. McRae, "A combinatorial optimization approach to molecular design," *Nanotechnology*, vol. 2, no. 3, p. 142, 1991.

[3] A. Sbihi and R. W. Eglese, "Combinatorial optimization and green logistics," *Annals of Operations Research*, vol. 175, no. 1, pp. 159–175, 2010.

[4] A. Schrijver and S.-V. (Berlin)., *Combinatorial Optimization: Polyhedra and Efficiency*. No. v. 1 in Algorithms and Combinatorics, Springer, 2003.

[5] M. Held and R. M. Karp, "A dynamic programming approach to sequencing problems," *Journal of the Society for Industrial and Applied mathematics*, vol. 10, no. 1, pp. 196–210, 1962.

[6] Gurobi Optimization, LLC, "Gurobi Optimizer Reference Manual," 2021.

[7] W. Cook, "Concorde," 2021.

[8] S. Lin, "Computer solutions of the traveling salesman problem," *Bell System Technical Journal*, vol. 44, no. 10, pp. 2245–2269, 1965.

[9] A. Blazinskas and A. Misevicius, "combining 2-opt, 3-opt and 4-opt with k-swap-kick perturbations for the traveling salesman problem," *Kaunas University of Technology, Department of Multimedia Engineering, Studentu St*, pp. 50–401, 2011.

[10] L. Perron and V. Furnon, "Or-tools."

[11] N. Christofides, "Worst-case analysis of a new heuristic for the travelling salesman problem," tech. rep., Carnegie-Mellon Univ Pittsburgh Pa Management Sciences Research Group, 1976.

[12] J. Jumper, R. Evans, A. Pritzel, T. Green, M. Figurnov, O. Ronneberger, K. Tunyasuvunakool, R. Bates, A. Žídek, A. Potapenko, *et al.*, "Highly accurate protein structure prediction with alphafold," *Nature*, pp. 1–11, 2021.

[13] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.

[14] O. Vinyals, M. Fortunato, and N. Jaitly, "Pointer networks," *Advances in Neural Information Processing Systems*, vol. 28, pp. 2692–2700, 2015.

[15] M. Nazari, A. Oroojlooy, M. Takáč, and L. V. Snyder, "Reinforcement learning for solving the vehicle routing problem," in *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, pp. 9861–9871, 2018.

[16] W. Kool, H. van Hoof, and M. Welling, "Attention, learn to solve routing problems!," in *International Conference on Learning Representations*, 2018.

[17] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio, "Graph attention networks," in *International Conference on Learning Representations*, 2018.

[18] H. Dai, E. B. Khalil, Y. Zhang, B. Dilkina, L. Song, and A. Financial, "Learning combinatorial optimization algorithms over graphs,"

[19] A. Nowak, S. Villar, A. S. Bandeira, and J. Bruna, "A note on learning algorithms for quadratic assignment with graph neural networks,"

[20] C. K. Joshi, T. Laurent, and X. Bresson, "An efficient graph convolutional network technique for the travelling salesman problem," *arXiv preprint arXiv:1906.01227*, 2019.

[21] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in neural information processing systems*, pp. 5998–6008, 2017.

[22] C. Joshi, "Transformers are graph neural networks," *The Gradient*, 2020.

[23] A. Dosovitskiy, L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly, *et al.*, "An image is worth 16x16 words: Transformers for image recognition at scale," *arXiv preprint arXiv:2010.11929*, 2020.

[24] Z. Liu, Y. Lin, Y. Cao, H. Hu, Y. Wei, Z. Zhang, S. Lin, and B. Guo, "Swin transformer: Hierarchical vision transformer using shifted windows," *arXiv preprint arXiv:2103.14030*, 2021.

[25] M. Deudon, P. Cournut, A. Lacoste, Y. Adulyasak, and L.-M. Rousseau, "Learning heuristics for the tsp by policy gradient," in *International conference on the integration of constraint programming, artificial intelligence, and operations research*, pp. 170–181, Springer, 2018.

[26] I. Bello, H. Pham, Q. Le, M. Norouzi, and S. Bengio, "Neural combinatorial optimization with reinforcement learning," 2017.

[27] Y. Kaempfer and L. Wolf, "Learning the multiple traveling salesmen problem with permutation invariant pooling networks," *arXiv preprint arXiv:1803.09621*, 2018.

[28] Y. Wu, W. Song, Z. Cao, J. Zhang, and A. Lim, "Learning improvement heuristics for solving routing problems..," *IEEE Transactions on Neural Networks and Learning Systems*, 2021.

[29] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, *et al.*, "Mastering the game of go without human knowledge," *nature*, vol. 550, no. 7676, pp. 354–359, 2017.

[30] K. Abe, Z. Xu, I. Sato, and M. Sugiyama, "Solving np-hard problems on graphs with extended alphago zero," *arXiv preprint arXiv:1905.11623*, 2019.

[31] Z. Xing and S. Tu, "A graph neural network assisted monte carlo tree search approach to traveling salesman problem," *IEEE Access*, vol. 8, pp. 108418–108428, 2020.

[32] A. Nash, K. Daniel, S. Koenig, and A. Felner, "Theta^*: Any-angle path planning on grids," in *AAAI*, vol. 7, pp. 1177–1183, 2007.

[33] D. Ferguson and A. Stentz, "Field d*: An interpolation-based path planner and replanner," in *Robotics research*, pp. 239–253, Springer, 2007.

[34] D. Dolgov, S. Thrun, M. Montemerlo, and J. Diebel, "Practical search techniques in path planning for autonomous driving," *Ann Arbor*, vol. 1001, no. 48105, pp. 18–80, 2008.

[35] A. H. Qureshi, A. Simeonov, M. J. Bency, and M. C. Yip, "Motion planning networks," in *2019 International Conference on Robotics and Automation (ICRA)*, pp. 2118–2124, IEEE, 2019.

[36] J. D. Gammell, S. S. Srinivasa, and T. D. Barfoot, "Informed rrt*: Optimal sampling-based path planning focused via direct sampling of an admissible ellipsoidal heuristic," in *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 2997–3004, IEEE, 2014.

[37] S. Wang, B. Z. Li, M. Khabsa, H. Fang, and H. Ma, "Linformer: Self-attention with linear complexity," *arXiv preprint arXiv:2006.04768*, 2020.

[38] Y. Xu, T. Fan, M. Xu, L. Zeng, and Y. Qiao, "Spidercnn: Deep learning on point sets with parameterized convolutional filters," 03 2018.

[39] H. Gao and S. Ji, "Graph u-nets," in *international conference on machine learning*, pp. 2083–2092, PMLR, 2019.

[40] J. Lee, I. Lee, and J. Kang, "Self-attention graph pooling," in *International conference on machine learning*, pp. 3734–3743, PMLR, 2019.

[41] P. C. Mahalanobis, "A sample survey of the acreage under jute in bengal," *Sankhyā: The Indian Journal of Statistics*, pp. 511–530, 1940.

[42] E. S. Marks, "A lower bound for the expected travel among m random points," *The Annals of Mathematical Statistics*, vol. 19, no. 3, pp. 419–422, 1948.

[43] M. Ghosh, "Expected travel among random points in a region," *Calcutta Statistical Association Bulletin*, vol. 2, no. 2, pp. 83–87, 1949.

[44] J. Beardwood, J. H. Halton, and J. M. Hammersley, "The shortest path through many points," in *Mathematical Proceedings of the Cambridge Philosophical Society*, vol. 55, pp. 299–327, Cambridge University Press, 1959.

A Proof

Sketch Proof of Proposition 1. Since a rigorous proof will include unnecessarily complicated details, we provide an intuitive sketch proof to this claim. One may refer to the works [41–44] for details.

An eligible TSP tour is a selection of n edges from a complete graph of n nodes randomly sampled from a unit square. The edges need to be connected head to tail and visit each node once and only once. For each node, if we select the edge that connects itself to its nearest neighbor, the sum of the costs of all such edges will obviously be less or equal to the optimal TSP cost $C(V_n)$, because choosing all nearest neighbors does not guarantee the selected edges will form an eligible TSP tour. We denote the cost of the edge between a node v to its nearest neighbor as $N(v)$. Thus,

$$\begin{aligned} \mathbb{E}(C(V_n)) &\geq \mathbb{E}\left(\sum_{i=1}^{n} N(v_i)\right) \\ &= \sum_{i=1}^{n} \mathbb{E}(N(v_i)) \\ &\approx n\,\mathbb{E}(N(v)) \qquad (v_i\text{'s are i.i.d.}) \end{aligned} \tag{4}$$

The distribution of $N(v)$ can be converted to an equivalent problem: what is the probability of the distance to the nearest neighbor of v being larger than a certain value r? Consider the area of the *intersection* of a circle centered at v of a radius r and the unit square, $S(r, v)$. For the distance to be larger than r, all the other $n-1$ points need to be outside of $S(r, v)$. Therefore,

$$P(N(v) > r) = (1 - S(r, v))^{n-1}, \tag{5}$$

and naturally, the CDF of $N(v)$ becomes $P(N(v) \leq r) = 1 - (1 - S(r, v))^{n-1}$. We could further get the PDF of $N(v)$ by taking the derivative of its CDF. However, the exact expression of $S(r, v)$ can be complicated when considering the boundary of the unit square. Fortunately, for sufficiently large n, $(1 - S(r, v))^{n-1}$ drops very fast, which means most probability mass is concentrated in a small circle around v. If we only consider the case where $S(r, v)$ is a circle and drops the remaining probability mass, we can roughly write

$$P(N(v) \leq r) \approx 1 - (1 - \pi r^2)^{n-1}. \tag{6}$$

Following this,

$$\begin{aligned}
\mathbb{E}(N(v)) &\approx \int_0^\epsilon r p_{N(v)}(r) dr \\
&= \int_0^\epsilon r \left(-(n-1)(1-\pi r^2)^{n-2}(-2\pi r)\right) dr \\
&= 2\pi(n-1) \int_0^\epsilon r^2 (1-\pi r^2)^{n-2} dr,
\end{aligned} \tag{7}$$

where ϵ denotes a small radius around v. From this point, it still can be challenging to directly calculate the integral, which will involve a hypergeometric function. However, given the fact that r is a small value, we can get a simpler analytical approximation by replacing $(1 - \pi r^2)^{n-2}$ with a close approximation $e^{-\pi(n-2)r^2}$. We also extend the integral boundary from ϵ to ∞, the difference of which should be negligible for a large n.

$$\begin{aligned}
\mathbb{E}(N(v)) &\approx 2\pi(n-1) \int_0^\epsilon r^2 e^{-\pi(n-2)r^2} dr \\
&\overset{\text{(large } n\text{)}}{\approx} 2\pi(n-1) \int_0^\infty r^2 e^{-\pi(n-2)r^2} dr \\
&= 2\pi(n-1) \cdot \frac{1}{4\pi(n-2)^{\frac{3}{2}}} \\
&= \frac{n-1}{2(n-2)^{\frac{3}{2}}} \\
&= O\left(\frac{1}{\sqrt{n}}\right).
\end{aligned} \tag{8}$$

Combining (4) and (8), $\mathbb{E}(C(V_n)) = O(\sqrt{n})$. □