Combinatorial Optimization by Graph Pointer Networks and Hierarchical Reinforcement Learning

Qiang Ma[1], **Suwen Ge**[1], **Danyang He**[1], **Darshan Thaker**[1], **Iddo Drori**[1,2]
[1]Columbia University [2]Cornell University
{ma.qiang, sg3635, dh2914, darshan.thaker}@columbia.edu
idrori@cs.columbia.edu

Abstract

In this work, we introduce Graph Pointer Networks (GPNs) trained using reinforcement learning (RL) for tackling the traveling salesman problem (TSP). GPNs build upon Pointer Networks by introducing a graph embedding layer on the input, which captures relationships between nodes. Furthermore, to approximate solutions to constrained combinatorial optimization problems such as the TSP with time windows, we train hierarchical GPNs (HGPNs) using RL, which learns a hierarchical policy to find an optimal city permutation under constraints. Each layer of the hierarchy is designed with a separate reward function, resulting in stable training. Our results demonstrate that GPNs trained on small-scale TSP50/100 problems generalize well to larger-scale TSP500/1000 problems, with shorter tour lengths and faster computational times. We verify that for constrained TSP problems such as the TSP with time windows, the feasible solutions found via hierarchical RL training outperform previous baselines. In the spirit of reproducible research we make our data, models, and code publicly available (Ma et al. 2019).

Introduction

Combinatorial optimization problems have received wide attention in the past few decades. One of the most important and practical problems is the traveling salesman problem (TSP). To introduce the TSP, consider a salesman who is traveling on a tour across a set of cities while minimizing the overall tour length. TSP is known to be an NP-hard problem (Papadimitriou 1977). Several approximation algorithms and heuristics for TSP have been proposed such as the 2-opt heuristic (Aarts, Aarts, and Lenstra 2003), Christofides algorithm (Christofides 1976), and the Lin-Kernighan heuristic (LKH) (Helsgaun 2000).

To solve combinatorial optimization problems, an increasing number of recent works leverage machine learning (ML) and reinforcement learning (RL) approach (Vinyals, Fortunato, and Jaitly 2015; Bello et al. 2017; Nazari et al. 2018; Khalil et al. 2017; Kool, van Hoof, and Welling 2019a; Joshi, Laurent, and Bresson 2019). A *seq2seq* model, known as the *pointer network* (Vinyals, Fortunato, and Jaitly 2015), has great potential in approximating solutions to several combinatorial optimization problems such as finding the convex hull and the TSP. An RL framework for pointer networks has been proposed (Bello et al. 2017), which outperformed most of the previous heuristics on TSP with up to 100 nodes. Recently, motivated by the Transformer architecture (Vaswani et al. 2017), Kool et al. proposed an attention model (Kool, van Hoof, and Welling 2019a; 2019b) to solve routing problems such as the TSP and VRP, significantly improving the result for small-scale TSP. However, scale is still an issue for the attention model.

As a powerful tool to capture graph information, Graph Neural Networks (GNNs) (Kipf and Welling 2016; Xu et al. 2019; Gasse et al. 2019) have been studied extensively and applied to solve combinatorial optimization problems. Li et al. (Li, Chen, and Koltun 2018) applied a Graph Convolutional Network (GCN) model (Kipf and Welling 2016) to solve several graph-based combinatorial optimization problems. Dai et al. (Khalil et al. 2017) proposed a graph embedding network trained with deep Q-learning and found that this generalized well to larger-scale problems. In addition to unconstrained problems, combinatorial optimization problems with constraints, e.g. TSP with time window (TSPTW), have not been fully considered. In this work, we explore the use of hierarchical RL methods (Kulkarni et al. 2016; Haarnoja et al. 2018a) to tackle combinatorial optimization problems with constraints, which are split into different subtasks. Each layer of the hierarchy learns to search the feasible solutions under constraints or learns the heuristics to optimize the objective function.

In this work, we approximate solutions to larger-scale TSP problems and address constrained combinatorial optimization problems. Our contributions are three-fold: (i) We propose a *graph pointer network* (GPN) to tackle the vanilla TSP. The GPN extends the pointer network with graph embedding layers and achieves faster convergence; (ii) We add a vector context to the GPN architecture and train using early stopping in order to generalize our model to tackle larger-scale TSP instances, e.g. TSP1000, from a model trained on a much smaller TSP50 instance; and (iii) We employ a hierarchical RL framework along with the GPN architecture to efficiently solve TSP with a time window constraint.

Copyright © 2020, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

Preliminaries

Traveling Salesman Problem

In this work, we focus on solving the symmetric 2-D Euclidean traveling salesman problem (TSP) (Lawler et al. 1985). The graph of the symmetric TSP is complete and undirected. Given a list of N city coordinates $\{\mathbf{x}_1, \mathbf{x}_2, ..., \mathbf{x}_N\} \subset \mathbb{R}^2$, we wish to find an optimal permutation σ over the cities that minimizes the tour length (Bello et al. 2017):

$$L(\sigma, \mathbf{X}) = \sum_{i=1}^{N} \|\mathbf{x}_{\sigma(i)} - \mathbf{x}_{\sigma(i+1)}\|_2, \tag{1}$$

where $\sigma(1) = \sigma(N+1)$, $\sigma(i) \in \{1, ..., N\}$, $\sigma(i) \neq \sigma(j)$ for any $i \neq j$, and $\mathbf{X} = [\mathbf{x}_1^\top, ..., \mathbf{x}_N^\top]^\top \in \mathbb{R}^{N \times 2}$ is a matrix consisting of all city coordinates $\mathbf{x}_i$.

Reinforcement Learning for TSP

We begin by introducing the notation used to formulate the TSP as a reinforcement learning problem. Let $\mathcal{S}$ be the state space and $\mathcal{A}$ be the action space. Each state $\mathbf{s}_t \in \mathcal{S}$ is defined as the set of all previous visited cities, i.e. $\mathbf{s}_t = \{\mathbf{x}_{\sigma(i)}\}_{i=1}^{t}$. The action $\mathbf{a}_t \in \mathcal{A}$ is defined as the next selected city, that is $\mathbf{a}_t = \mathbf{x}_{\sigma(t+1)}$. Since $\sigma(1) = \sigma(N+1)$, it follows that $\mathbf{a}_N = \mathbf{x}_{\sigma(N+1)} = \mathbf{x}_{\sigma(1)}$, which means the last choice of the route is the start city.

Denote a policy as $\pi_\theta(\mathbf{a}_t|\mathbf{s}_t)$, which is a distribution over candidate cities $\mathbf{a}_t$ given a set of visited cities $\mathbf{s}_t$. Given a set of visited cities, the policy will return a probability distribution over the next candidate cities that have not been chosen. In our case, the policy is represented by a neural network and the parameter θ represents the trainable weights of the neural network. Furthermore, the reward function is defined as the negative cost incurred from taking action a_t from state s_t, i.e. $r(\mathbf{s}_t, \mathbf{a}_t) = -\|\mathbf{x}_{\sigma(t)} - \mathbf{x}_{\sigma(t+1)}\|_2$. Then the expected reward (Sutton and Barto 2018) is defined as follows:

$$\begin{aligned} &\mathbb{E}_{(\mathbf{s}_t,\mathbf{a}_t) \sim \pi_\theta(\mathbf{s}_t,\mathbf{a}_t)} \left[\sum_{t=1}^{N} r(\mathbf{s}_t, \mathbf{a}_t) \right] \\ &= \mathbb{E}_{\sigma \sim p_\theta(\cdot), \mathbf{X} \sim \mathcal{X}} \left[\sum_{t=1}^{N} -\|\mathbf{x}_{\sigma(t)} - \mathbf{x}_{\sigma(t+1)}\|_2 \right] \end{aligned} \tag{2}$$

where $\mathcal{X}$ is the space of the set of cities, Γ is the space of all possible permutations σ over $\mathcal{X}$, and $p_\theta(\Gamma)$ is the distribution over Γ, which is predicted by the neural network. To maximize the above reward function, the network must learn a policy to minimize the expected tour length. We employ the policy gradient algorithm (Sutton and Barto 2018) to learn to maximize the reward function as described next.

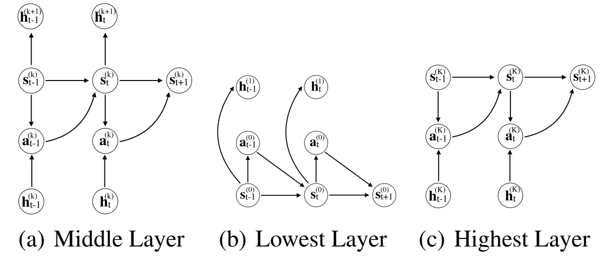
Hierarchical Reinforcement Learning

Hierarchical RL for TSP

A key aspect of our work is tackling TSP with constraints. Augmenting traditional RL reward functions with a penalty term encourages solutions to be in the feasible set (Bello et al. 2017); however, we find this method leads to unstable training. Instead, we propose a hierarchical RL framework to more efficiently tackle TSP with constraints.

Motivated by the work of Haarnoja et al. (Haarnoja et al. 2018a; 2018b), we adopt a probabilistic graphical model framework for control, as demonstrated in Figure 1. Each layer of a hierarchy defines a policy, from which we sample actions. At a given layer $k \in \{0, \dots, K\}$, the current action $\mathbf{a}_t^{(k)}$ is sampled from the policy $\pi_{\theta_k}(\mathbf{a}_t^{(k)}|\mathbf{s}_t^{(k)}, \mathbf{h}_t^{(k)})$, where $\mathbf{h}_t^{(k)} \in \mathcal{H}^{(k)}$ is a latent variable from the previous layer in the hierarchy and $\mathcal{H}^{(k)}$ is its corresponding latent space. For convenience of notation, on the k-th layer, we extend the policy to both sample the action and provide the latent variable, i.e. $\mathbf{a}_t^{(k)}, \mathbf{h}_t^{(k+1)} \sim \pi_{\theta_k}(\cdot|\mathbf{s}_t^{(k)}, \mathbf{h}_t^{(k)})$.

Each layer corresponds to a different RL task, so the reward functions are hand-designed to be different for each layer. For our experiments, we set lower layer reward functions to simply bias solutions to be in the feasible set of the constrained optimization problem, and set higher layer reward functions to be the original optimization objective. Hence the lower layer learns to search feasible solutions under constraints and provide prior for higher layer. This hierarchy yields fast convergence and better results.



(a) Middle Layer (b) Lowest Layer (c) Highest Layer

Figure 1: Graphical models for hierarchical RL framework.

Hierarchical Policy Gradient

We use the policy gradient method to learn a hierarchical policy. Considering a hierarchical policy, the objective function of the k-th layer is $J(\theta_k) = -\mathbb{E}_{\sigma \sim p_{\theta_k(\sigma)}, \mathbf{X} \sim \mathcal{X}}[L(\sigma, \mathbf{X})]$. Based on the REINFORCE algorithm, the gradient of the k-th layer policy is expressed as (Williams 1992; Bello et al. 2017):

$$\begin{aligned} \nabla_{\theta_k} J(\theta_k) = \frac{1}{B} \sum_{i=1}^{B} \Bigg[\left(\sum_{t=1}^{N} r_k(\mathbf{s}_{i,t}^{(k)}, \mathbf{a}_{i,t}^{(k)}) - b_{i,k} \right) \\ \times \left(\sum_{t=1}^{N} \nabla_{\theta_k} \log \pi_{\theta_k}(\mathbf{a}_{i,t}^{(k)}|\mathbf{s}_{i,t}^{(k)}, \mathbf{h}_{i,t}^{(k)}) \right) \Bigg], \end{aligned} \tag{3}$$

where B is the batch size, π_{θ_k} is the k-th layer policy, $r_k(\cdot, \cdot)$ is the reward function for the k-th layer, $b_{i,k}$ is the k-th layer *central self-critic* baseline, and $\mathbf{h}_t^{(k)}$ is the latent variable from the lower layer. The central self-critic baseline $b_{i,k}$ is

defined as:

$$b_{i,k} = \sum_{t=1}^{N} \left(r_k(\tilde{\mathbf{s}}_{i,t}^{(k)}, \tilde{\mathbf{a}}_{i,t}^{(k)}) \right) + \left[\frac{1}{B} \sum_{t=1}^{N} \sum_{j=1}^{B} \left(r_k(\mathbf{s}_{j,t}^{(k)}, \mathbf{a}_{j,t}^{(k)}) - r_k(\tilde{\mathbf{s}}_{j,t}^{(k)}, \tilde{\mathbf{a}}_{j,t}^{(k)}) \right) \right] \quad (4)$$

Based on Equation 3, θ_k are optimized using gradient descent through the update rule $\theta_k \leftarrow \theta_k + \alpha \nabla_{\theta_k} J(\theta_k)$.

Layer-Wise Policy Optimization

Suppose we need to learn a $(K+1)$-layer hierarchical policy, which includes $\pi_{\theta_0}, \pi_{\theta_1}, ..., \pi_{\theta_K}$. Each policy is represented by a GPN. In order to learn policy π_{θ_K}, we first need to train all lower layers π_{θ_k} for $k = 0, ..., K-1$ and fix the weights of the neural networks. Then, for layer $k = 0, ..., K-1$, we sample $(\mathbf{s}_t^{(k)}, \mathbf{a}_t^{(k)})$ based on π_{θ_k}, and provide latent variable $\mathbf{h}_t^{(k+1)}$ for the next higher layer. Finally, we can learn the policy π_{θ_K} from $\mathbf{h}_t^{(K)}$. Algorithm 1 provides detailed pseudo-code.

Algorithm 1 Layer-Wise Policy Optimization

```
1: procedure TRAIN(training set 𝒳, # of training steps M_0, M_1, ..., M_K, batch size B, learning rate α, the number of layers K)
2:     Initialize network parameters θ_k for k ∈ {0, ..., K}
3:     for k = 0 to K do
4:         for m = 1 to M_k do
5:             X_i ~Sample(𝒳) for i ∈ {1, ..., B}
6:             for j = 0 to k − 1 do
7:                 a_{i,t}^{(j)}, h_{i,t}^{(j+1)} ~ π_{θ_j}(·|s_{i,t}^{(j)}, h_{i,t}^{(j)})
8:             a_{i,t}^{(k)} ~ π_{θ_k}(·|s_{i,t}^{(k)}, h_{i,t}^{(k)})
9:             ã_{i,t}^{(k)} ~ π_{θ_k}^{Greedy}(·|s̃_{i,t}^{(k)}, h_{i,t}^{(k)})
10:            Compute J(θ_k), ∇_{θ_k} J(θ_k)
11:            θ_k ← θ_k + α∇_{θ_k} J(θ_k)
12:    return π_{θ_0}, π_{θ_1}, ..., π_{θ_K}
```

Graph Pointer Network

GPN Architecture

We propose a *graph pointer network* (GPN) based on the pointer network (Bello et al. 2017) for approximately solving the TSP. The GPN architecture, which is shown in Figure 2, consists of an encoder and decoder component.

Encoder The encoder includes two parts: point encoder and graph encoder. For the point encoder, each city coordinate $\mathbf{x}_i$ is embedded into a higher dimensional vector $\tilde{\mathbf{x}}_i \in \mathbb{R}^d$, where d is the hidden dimension. The vector $\tilde{\mathbf{x}}_i$ for the current city $\mathbf{x}_i$ is then encoded by an LSTM. The hidden variable $\mathbf{x}_i^h$ of the LSTM is passed to both the decoder and the encoder in the next time step. For the graph encoder, we use graph embedding layers to encode all city coordinates $\mathbf{X} = [\mathbf{x}_1^\top, ..., \mathbf{x}_N^\top]^\top$, and pass it to the decoder.

Graph Embedding Layer In TSP, the context information of a city node includes the neighbors' information of the city. In a GPN, context information is obtained by encoding all city coordinates $\mathbf{X}$ via a graph neural network (GNN) (Kipf and Welling 2016; Xu et al. 2019). Each layer of the GNN is expressed as:

$$\mathbf{x}_i^l = \gamma \mathbf{x}_i^{l-1} \Theta + (1-\gamma) \phi_\theta \left(\frac{1}{|\mathcal{N}(i)|} \{\mathbf{x}_j^{l-1}\}_{j \in \mathcal{N}(i) \cup \{i\}} \right), \quad (5)$$

where $\mathbf{x}_i^l \in \mathbb{R}^{d_l}$ is the l-th layer variable with $l \in \{1, ..., L\}$, $\mathbf{x}_i^0 = \mathbf{x}_i$, γ is a trainable parameter which regularizes the eigenvalue of the weight matrix, $\Theta \in \mathbb{R}^{d_{l-1} \times d_l}$ is a trainable weight matrix, $\mathcal{N}(i)$ is the adjacency set of node i, and $\phi_\theta : \mathbb{R}^{d_{l-1}} \rightarrow \mathbb{R}^{d_l}$ is the aggregation function (Kipf and Welling 2016), which is represented by a neural network. Furthermore, the graph embedding layer is further expressed as $\mathbf{X}^l = \gamma \mathbf{X}^{l-1} \Theta + (1-\gamma) \Phi_\theta \left(\mathbf{X}^{l-1} / |\mathcal{N}(i)| \right)$, where $\mathbf{X}^l \in \mathbb{R}^{N \times d_l}$, and $\Phi_\theta : \mathbb{R}^{N \times d_{l-1}} \rightarrow \mathbb{R}^{N \times d_l}$ is the aggregation function.

Vector Context In previous work (Bello et al. 2017), the context is computed based on the 2D coordinates of all cities, i.e. $\mathbf{X} \in \mathbb{R}^{N \times 2}$. We refer to this context as *point context*. In contrast, instead of using coordinate features directly, in this work, we use the vectors pointing from the current city to all other cities as the context, which we refer to as a *vector context*. For the current city $\mathbf{x}_i$, suppose $\mathbf{X}_i = [\mathbf{x}_i^\top, ..., \mathbf{x}_i^\top]^\top \in \mathbb{R}^{N \times 2}$ is a matrix with identical rows $\mathbf{x}_i$. We define $\bar{\mathbf{X}}_i = \mathbf{X} - \mathbf{X}_i$ as the vector context. The j-th row of $\bar{\mathbf{X}}_i$ is a vector pointing from node i to node j. Then $\bar{\mathbf{X}}_i$ is passed into the graph embedding layers.

Decoder The decoder is based on an attention mechanism and outputs the pointer vector $\mathbf{u}_i$, which is then passed to a softmax layer to generate a distribution over the next candidate cities. Similar to pointer networks (Bello et al. 2017), the attention mechanism and pointer vector $\mathbf{u}_i$ is defined as:

$$\mathbf{u}_i^{(j)} = \begin{cases} v^\top \cdot \tanh(W_r r_j + W_q q) & \text{if } j \neq \sigma(k), \forall k < j, \\ -\infty & \text{otherwise,} \end{cases} \quad (6)$$

where $\mathbf{u}_i^{(j)}$ is the j-th entry of the vector $\mathbf{u}_i$, W_r and W_q are trainable matrices, q is a query vector from the hidden variable of the LSTM, and r_i is a reference vector containing the information of the context of all cities. Precisely, we use the hidden variable $\mathbf{x}_i^h$ from the point encoder as the query vector q, and use the context $\mathbf{X}^L$ from the graph embedding layer as the reference, i.e. $q = \mathbf{x}_i^h$ and $r_j = \mathbf{X}_j^L$. The distribution policy over all candidate cities is given by $\pi_\theta(\mathbf{a}_i|\mathbf{s}_i) = \mathbf{p}_i = \text{softmax}(\mathbf{u}_i)$. We predict the next visited city $\mathbf{a}_i = \mathbf{x}_{\sigma(i+1)}$, by sampling or choosing greedily from the policy $\pi_\theta(\mathbf{a}_i|\mathbf{s}_i)$.

Hierarchical GPN Architecture

In this section, we use the proposed GPN to design a hierarchical architecture. The architecture of a two-layer hierarchical GPN (HGPN) is illustrated in Figure 3. In contrast to a single-layer GPN, the coordinate $\mathbf{x}_i^{(k)}$ at k-th layer

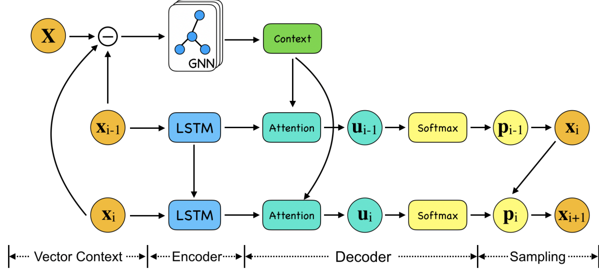


Figure 2: Architecture for Graph Pointer Network.

is first passed as input to a lower-level neural network and the network outputs a pointer vector $\mathbf{u}_i^{(k-1)}$. Then, $\mathbf{u}_i^{(k-1)}$ is added to the pointer vector $\mathbf{u}_i^{(k)}$ of a higher layer, i.e. $\mathbf{p}_i^{(k)} = \text{softmax}(\mathbf{u}_i^{(k)} + \alpha \mathbf{u}_i^{(k-1)})$, where α is a trainable parameter. This plays an important role since $\mathbf{u}_i^{(k-1)}$ contains lower layer information which provides a prior distribution over the output cities. The output $\mathbf{x}_{i+1}^{(k)}$ is then sampled from $\pi_\theta(\cdot|\mathbf{s}_i^{(k)}, \mathbf{h}_i^{(k)}) = \mathbf{p}_i^{(k)}$, where $\mathbf{h}_i^{(k)} = \mathbf{u}_i^{(k-1)}$ is the latent variable from the lower layer.

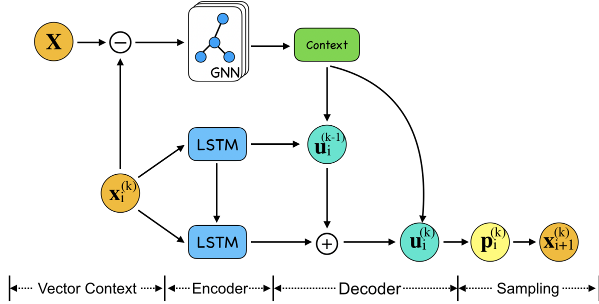


Figure 3: A two layer hierarchical architecture of GPN.

Experiments

In this section, we provide more experiments and implementation details for the experiments. We use $L = 3$ graph embedding layers to encode the context in the GPN. The aggregation function used is a single layer fully connected neural network. The graph embedding layer is expressed as

$$\mathbf{X}^l = \gamma \mathbf{X}^{l-1} \Theta + (1-\gamma) g(\mathbf{X}^{l-1} W / |\mathcal{N}(i)| + b), \quad (7)$$

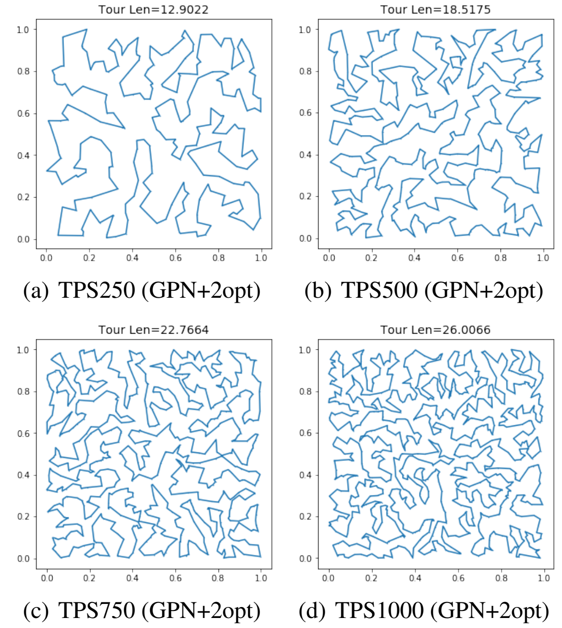
where $g(\cdot)$ is the ReLU activation function, $W \in \mathbb{R}^{d_{l-1} \times d_a}$ and $b \in \mathbb{R}^{N \times d_a}$ are trainable weights and biases with $d_l = d_a = 128$ for $l = 1, 2, 3$. The training data is generated randomly from a $[0,1]^2$ uniform distribution. In each epoch, the training data is generated on the fly.

Experiments for larger-scale TSP

In real world applications, most practical TSP instances have hundreds or thousands of nodes. We find that the proposed GPN model generalizes well from small-scale TSP problems to larger scale. The generalization capacity increases by an order of magnitude.

In Table 1, we train a GPN model with vector context on TSP50 data with 10 epochs, and use this model to predict the routes on TSP250/500/750/1000. Furthermore, we use a local search algorithm 2-opt to improve our results after prediction. The baseline models Pointer Network (PN) (Bello et al. 2017), s2v-DQN (Khalil et al. 2017) and Attention Model (AM) (Kool, van Hoof, and Welling 2019a) are also trained with TSP50 data. Results are averaged over 1000 TSP instances. The results are also compared with LKH, Concorde, nearest neighbor, 2-opt, farthest insertion and Google OR-Tools (Google 2016).

Table 1 shows that our GPN model outperforms PN and AM when we train with TSP50 instances and generalize to larger-scale problems. With local search added, the GPN+2opt has similar tour length to s2v-DQN, but saves $\approx 20\%$ running time. Compared with the 2-opt heuristic, the GPN+2opt uses $\approx 25\%$ less running time, which means the GPN model can be treated as a good initialization method. The GPN+2opt also outperforms OR-Tools on TSP1000. On Table 1, GPN does not outperform the state-of-the-art TSP solver, e.g. LKH and Farthest Insertion. However, it still has the potential to be an effective initialization method, since the GPN shows good generalization capabilities and can solve TSP instances in parallel. Some sample tours are shown in Figure 4.



(a) TPS250 (GPN+2opt) (b) TPS500 (GPN+2opt)

(c) TPS750 (GPN+2opt) (d) TPS1000 (GPN+2opt)

Figure 4: Sample tours for TSP250/500/750/1000. Approximate solutions of larger-scale TSP predicted by GPN and 2-opt heuristics.

As aforementioned, the generalization capacity of the GPN model is roughly an order of magnitude larger than the size of the instances the model is trained on. More specifically, we train the GPN models on TSP20/50/100 and use

Table 1: Comparison for larger-scale TSP. Each result is obtained by running on 1000 random TSP instances. Tour Len refers to average tour length. Time refers to total running time (sec) of 1000 instances.

Method	TSP 250		TSP 500		TSP 750		TSP 1000	
	Tour Len.	Time	Tour Len.	Time	Tour Len.	Time	Tour Len.	Time
LKH	11.893	9792s	16.542	23070s	20.129	36840s	23.130	50680s
Concorde	11.89	1894s	16.55	13902s	20.10	32993s	23.11	47804s
Nearest Neighbor	14.928	25s	20.791	60s	25.219	115s	28.973	136s
2-opt	13.253	303s	18.600	1363s	22.668	3296s	26.111	6153s
Farthest Insertion	13.026	33s	18.288	160s	22.342	454s	25.741	945s
OR-Tools (Savings)	12.652	5000s	17.653	5000s	22.933	5000s	28.332	5000s
OR-Tools (Christofides)	12.289	5000s	17.449	5000s	22.395	5000s	26.477	5000s
s2v-DQN	13.079	476s	18.428	1508s	22.550	3182s	26.046	5600s
Pointer Net	14.249	29s	21.409	280s	27.382	782s	32.714	3133s
Attention Model	14.032	2s	24.789	14s	28.281	42s	34.055	136s
GPN (ours)	13.679	32s	19.605	111s	24.337	232s	28.471	393s
GPN+2opt (ours)	12.942	214s	18.358	974s	22.541	2278s	26.129	4410s

these models to predict on TSP500/1000. The results are shown in Table 2, which demonstrates that the results improve if we increase the size of the training TSP instances.

Table 2: Comparison for larger-scale TSP. The GPNs are trained with different size of TSP instances and generalize on larger-scale problems.

Model	TSP 500		TSP 1000	
	Tour Len.	Time	Tour Len.	Time
GPN (TSP20)	22.320	107s	33.649	391s
GPN (TSP50)	19.605	111s	28.471	393s
GPN (TSP100)	**19.527**	109s	**28.036**	408s

Experiments for TSP with time window

We consider a well known constrained TSP problem, the TSP with Time Windows (TSPTW). In TSPTW, each node i has its own service time interval $[e_i, l_i]$, where e_i is the entering time and l_i is the leaving time. A city cannot be visited after its leaving time. In this experiment, we consider the following formalization of the TSPTW problem:

$$\begin{aligned} \min_{\sigma} \quad & c_N \\ \text{s.t.} \quad & c_{i+1} - c_i \geq \|\mathbf{x}_{\sigma(i+1)} - \mathbf{x}_{\sigma(i)}\|_2, \quad & i \in \{1, ..., N-1\}, \\ & e_i \leq c_i \leq l_i & i \in \{1, ..., N\}, \end{aligned}$$

where c_i is the arriving time for the i-th city. To tackle the TSPTW problem, we construct a two-layer hierarchical GPN (HGPN) trained on TSPTW20 instances.

At prediction time, we use both the greedy and sampling method. The result is improved by sampling 100 or 500 times. Table 3 demonstrates that our HGPN framework outperforms all other baselines on TSPTW including the single-layer GPN. All instances have feasible solutions based on our training setup; however the algorithms may occasionally fail to find a feasible solution, which we quantify by using the percentage of feasible solutions as an evaluation metric. The HGPN achieves a much higher percentage of feasible solutions compared to the baselines. Figure 5 shows sample tours.

Table 3: Results for TSPTW20. Cost: objective of TSPTW. Time: prediction time. Feasible %: the percentage of instances that are predicted to have feasible solutions.

Method	Cost	Time	Feasible %
OR-Tools (Savings)	4.045	121s	72.06%
ACO	4.655	204s	62.10%
GPN-greedy	4.209	1s	99.87%
HGPN-greedy	4.178	1s	99.88%
HGPN-sampling-100	4.013	99s	100%
HGPN-sampling-500	**3.991**	494s	100%

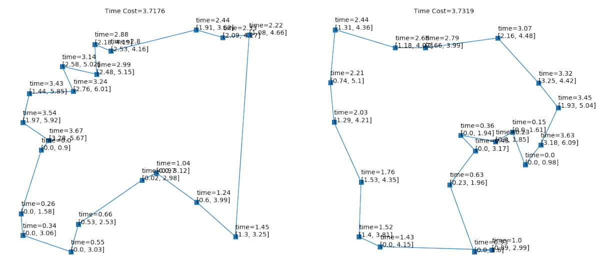


Figure 5: Sample tours for TSPTW20. For the text on each node, the first line is the arriving time, and the second line is the time window.

Conclusion

In this work, we propose a Graph Pointer Network (GPN) framework which efficiently solves larger-scale TSP by using graph neural networks. Training a hierarchical RL model allows our approach to additionally tackle constrained combinatorial optimization problems such as the TSPTW. Our experimental results demonstrate that the GPN generalizes well from small-scale to larger-scale problems, outperforming previous RL methods for combinatorial optimization. In the spirit of reproducible research, we make our data, models, and code publicly available (Ma et al. 2019).

References

Aarts, E.; Aarts, E. H.; and Lenstra, J. K. 2003. *Local search in combinatorial optimization*. Princeton University Press.

Bello, I.; Pham, H.; Le, Q. V.; Norouzi, M.; and Bengio, S. 2017. Neural combinatorial optimization with reinforcement learning. *International Conference on Learning Representations Workshop*.

Christofides, N. 1976. Worst-case analysis of a new heuristic for the travelling salesman problem. Technical report, Carnegie-Mellon University, Pittsburgh Management Sciences Research Group.

Gasse, M.; Chételat, D.; Ferroni, N.; Charlin, L.; and Lodi, A. 2019. Exact combinatorial optimization with graph convolutional neural networks. *Advances in Neural Information Processing Systems*.

Google. 2016. Or-tools, Google optimization tools. *https://developers.google.com/optimization/routing*.

Haarnoja, T.; Hartikainen, K.; Abbeel, P.; and Levine, S. 2018a. Latent space policies for hierarchical reinforcement learning. In *International Conference on Machine Learning*, 1846–1855.

Haarnoja, T.; Zhou, A.; Abbeel, P.; and Levine, S. 2018b. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *International Conference on Machine Learning*, 1856–1865.

Helsgaun, K. 2000. An effective implementation of the lin–kernighan traveling salesman heuristic. *European Journal of Operational Research* 126(1):106–130.

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

Khalil, E.; Dai, H.; Zhang, Y.; Dilkina, B.; and Song, L. 2017. Learning combinatorial optimization algorithms over graphs. In *Advances in Neural Information Processing Systems*, 6348–6358.

Kipf, T. N., and Welling, M. 2016. Semi-supervised classification with graph convolutional networks. *International Conference on Learning Representations*.

Kool, W.; van Hoof, H.; and Welling, M. 2019a. Attention, learn to solve routing problems! *International Conference on Learning Representations*.

Kool, W.; van Hoof, H.; and Welling, M. 2019b. Buy 4 reinforce samples, get a baseline for free! *International Conference on Learning Representations*.

Kulkarni, T. D.; Narasimhan, K.; Saeedi, A.; and Tenenbaum, J. 2016. Hierarchical deep reinforcement learning: Integrating temporal abstraction and intrinsic motivation. In *Advances in Neural Information Processing Systems*, 3675–3683.

Lawler, E. L.; Lenstra, J. K.; Kan, A. R.; Shmoys, D. B.; et al. 1985. *The traveling salesman problem: a guided tour of combinatorial optimization*, volume 3. Wiley New York.

Li, Z.; Chen, Q.; and Koltun, V. 2018. Combinatorial optimization with graph convolutional networks and guided tree search. In *Advances in Neural Information Processing Systems*, 537–546.

Ma, Q.; Ge, S.; He, D.; Thaker, D.; and Drori, I. 2019. GitHub Repository for Combinatorial Optimization by Graph Pointer Networksand Hierarchical Reinforcement Learning. *https://github.com/qiang-ma/graph-pointer-network*.

Nazari, M.; Oroojlooy, A.; Snyder, L.; and Takác, M. 2018. Reinforcement learning for solving the vehicle routing problem. In *Advances in Neural Information Processing Systems*, 9839–9849.

Papadimitriou, C. H. 1977. The euclidean travelling salesman problem is np-complete. *Theoretical Computer Science* 4(3):237–244.

Sutton, R. S., and Barto, A. G. 2018. *Reinforcement learning: An introduction*. MIT Press.

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

Vinyals, O.; Fortunato, M.; and Jaitly, N. 2015. Pointer networks. In *Advances in Neural Information Processing Systems*, 2692–2700.

Williams, R. J. 1992. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning* 8(3-4):229–256.

Xu, K.; Hu, W.; Leskovec, J.; and Jegelka, S. 2019. How powerful are graph neural networks? *International Conference on Learning Representations*.