Generalize a Small Pre-trained Model to Arbitrarily Large TSP Instances

Zhang-Hua Fu [1,2], **Kai-Bin Qiu** [2], **Hongyuan Zha** [1,3 *]

[1] Shenzhen Institute of Artificial Intelligence and Robotics for Society, Shenzhen, China
[2] Institute of Robotics and Intelligent Manufacturing, The Chinese University of Hong Kong, Shenzhen, China
[3] School of Data Science, The Chinese University of Hong Kong, Shenzhen, China
fuzhanghua@cuhk.edu.cn, 220019002@link.cuhk.edu.cn, zhahy@cuhk.edu.cn

Abstract

For the traveling salesman problem (TSP), the existing supervised learning based algorithms suffer seriously from the lack of generalization ability. To overcome this drawback, this paper tries to train (in supervised manner) a small-scale model, which could be repetitively used to build heat maps for TSP instances of arbitrarily large size, based on a series of techniques such as graph sampling, graph converting and heat maps merging. Furthermore, the heat maps are fed into a reinforcement learning approach (Monte Carlo tree search), to guide the search of high-quality solutions. Experimental results based on a large number of instances (with up to 10,000 vertices) show that, this new approach clearly outperforms the existing machine learning based TSP algorithms, and significantly improves the generalization ability of the trained model.

Introduction

The travelling salesman problem (TSP) is a well-known combinatorial optimization problem with various real-life applications, such as transportation, robots routing, biology, circuit design. Given n cities as well as the distance d_{ij} between each pair of cities i and j, the TSP aims to find a cheapest tour which starts from a beginning city (arbitrarily chosen), visits each city exactly once, and finally returns to the beginning city. This problem is NP-hard, thus being extremely difficult from the viewpoint of theoretical computer science.

Due to its importance in both theory and practice, many algorithms have been developed, mostly based on traditional operations research (OR) methods. Among the existing TSP algorithms, the best exact solver Concorde (Applegate et al. 2009) succeeded in demonstrating optimality of an Euclidean TSP instance with 85,900 cities, while the leading heuristics (Helsgaun 2017) and (Taillard and Helsgaun 2019) are capable of obtaining near-optimal solutions for instances with millions of cities. However, these algorithms are very complicated, which consist of many hand-crafted rules and heavily rely on expert knowledge, thus being difficult to generalize to other combinatorial optimization problems.

To overcome those limitations, recent years have seen a number of machine learning (ML) based algorithms being proposed for the TSP (briefly reviewed in the next section), which attempt to automate the search process by learning mechanisms. This type of methods do not rely heavily on expert knowledge, can be easily generalized to various combinatorial optimization problems, thus become a promising research direction.

For the TSP, existing ML based algorithms can be roughly classified into two categories, i.e., (1) supervised learning (SL) algorithms which attempt to discover common patterns supervised by pre-computed TSP solutions. (2) reinforcement learning (RL) algorithms which try to learn during the interaction with the environment (without pre-computed solutions).

Once well trained, SL models are able to provide useful information that significantly speeds up the search of high-quality TSP solutions. However, the performance of a pre-trained model of fixed size may decrease drastically while tackling TSP instances of different sizes, since the distributions of the training instances are very different from the test instances. On the other hand, training SL models generally requires a large number of pre-computed optimal (at least high-quality) TSP solutions, being unaffordable for large-scale TSP instances. These drawbacks seriously limit the usage of SL on large-scale TSP instances.

However, we believe the idea of discovering common patterns in a supervised manner is valuable. If we can train a small-scale SL model within reasonable time and find a way to smoothly generalize it to large-scale cases (without pre-computing a large number of solutions again), it is hopeful to inherit the advantages of SL while avoiding its drawbacks. Motivated by this idea, we develop a series of techniques, in order to improve the generalization ability of the model trained by SL. Furthermore, we combine SL and RL to form a hybrid algorithm, which performs favorably with respect to the existing ML based TSP algorithms. Overall, the main contributions are summarized as follows.

- **Methodologies**: At first, we train a small-scale (with size m) model by supervised learning, based on a graph convolutional residual network with attention mechanism (Att-GCRN). Once well trained, given a TSP instance with m vertices, the model is able to build a heat map over the edges. Then, we try to smoothly generalize this model to handle large instances. For this purpose, given a large-scale TSP instance, we repeatedly use a graph sampling method

*Corresponding author.

to extract a sub-graph with exactly m vertices, then convert it to a standard TSP instance, and call the pre-trained model to build a sub heat map. Finally, all the sub heat maps are merged together, to get a complete heat map over the original graph. Although the Att-GCRN is somewhat similar to the network in (Joshi, Laurent, and Bresson 2019), to our best knowledge, the graph sampling, graph converting and heat maps merging techniques are firstly developed for the TSP in this paper, which significantly improve the generalization ability of the trained model.

Furthermore, based on the merged heat map, we use a RL based approach, i.e., Monte Carlo tree search (MCTS), to search high-quality solutions. To our best knowledge, there are two existing works (Shimomura and Takashima 2016) and (Xing and Tu 2020) which also use MCTS to solve the TSP. However, they are both constructive approaches, where each state is a partial TSP tour, and each action adds a city to increase the partial tour. By contrast, our MCTS method is a conversion based approach, where each state is a complete tour, and each action converts the current state to a new complete tour. Therefore, our method is very different from the existing MCTS algorithms.

- **Results**: We carry out experiments on a large number of TSP instances with up to 10,000 cities (one order of magnitude larger than the instances used to evaluate the existing ML algorithms). On all the data sets, our new algorithm is able to obtain optimal or near-optimal solutions within reasonable time, clearly outperforming all the existing learning based algorithms.

Related Works

In this section, we briefly review the existing ML based algorithms on the TSP, and then extend to several other highly related problems. Non-learned methods are omitted, interested readers please find in (Applegate et al. 2009), (Rego et al. 2011), (Helsgaun 2017) and (Taillard and Helsgaun 2019) for an overlook of the leading TSP algorithms.

The idea of applying ML to solve the TSP dated back to several decades ago (Hopfield and Tank 1985), but becomes a hot and promising topic only in recent years. A number of ML based TSP algorithms have been developed, which can be classified into two categories.

Supervised learning (SL) methods: Vinyals, Fortunato, and Jaitly (2015) introduced a pointer network which consists of an encoder and a decoder, both using recurrent neural network (RNN). The encoder parses each TSP city into an embedding, and then the decoder uses an attention model to predict the probability distribution over the candidate (unvisited) cities. Nowak et al. (2017) proposed a supervised approach, which trains a graph neural network (GNN) to predict an adjacency matrix (heat map) over the cities, and then attempts to convert the adjacency matrix to a feasible TSP tour by beam search (OR based method). Joshi, Laurent, and Bresson (2019) followed this framework, but chose deep graph convolutional networks (GCN) to build heat map, and then constructed tours via highly parallelized beam search. Xing and Tu (2020) trained a graph neural network (GNN) to capture the local and global graph structure, based on which they used a MCTS procedure to construct TSP tours. These SL based methods require a large number of pre-computed TSP solutions, thus being difficult to directly generalize to large-scale instances.

Reinforcement learning (RL) methods: To overcome the drawback of SL, several groups chose RL instead of SL. For example, Bello et al. (2017) implemented an actor-critic RL architecture, which uses the tour length as a reward, to guide the search towards promising area. Khalil et al. (2017) proposed a framework which maintains a partial tour and repeatedly calls a RL model to select the most relevant city to add to the partial tour, until forming a complete TSP tour. Emami and Ranka (2018) also implemented an actor-critic neural network, and chose Sinkhorn policy gradient to learn policies by approximating a double stochastic matrix. Concurrently, (Deudon et al. 2018), (Kool, van Hoof, and Welling 2019) both proposed a graph attention network (GAN), which incorporates attention mechanism with RL to auto-regressively improve the quality of the obtained solution. Recently, Wu et al. (2020) presented an improvement based learning framework, which exploited deep RL to automatically discover better improvement policies.

In addition, there are several ML based methods recently proposed for other related problems, such as the decision TSP (Prates et al. 2019), the multiple TSP (Kaempfer and Wolf 2019), and the vehicle routing problem (Nazari et al. 2018), (Chen and Tian 2019) and (Lu, Zhang, and Yang 2020), etc. For an overall survey, please refer to (Bengio, Lodi, and Prouvost 2018) and (Guo et al. 2019).

Methods

Preliminaries

In this paper, we focus on the two-dimensional Euclidean TSP, which is formulated as an undirected graph $G(V, E)$, where V (with $|V| = n$) denotes the set of vertices (each vertex corresponds to a city), and E denotes the set of edges. Without loss of generality, assume all the vertices are distributed within a two dimensional unit square, i.e., for each vertex $i \in V$, its coordinates x_i and y_i both belong to $[0, 1]$, and the distance d_{ij} is defined as the Euclidean distance between vertices i and j. Furthermore, corresponding to graph G, its heat map is defined as a $n \times n$ matrix $\boldsymbol{P}$, whose element $P_{ij} \in [0, 1]$ denotes the probability of edge (i, j) belonging to the optimal TSP solution.

As a preliminary step, we at first train (off-line learning) a graph convolutional residual neural network with attention mechanisms (denoted by Att-GCRN for short, whose architecture is described in the full version of this paper [1]), with fixed input size m (a parameter). To train the model, 990,000 TSP instances with m vertices are randomly generated as the train set, and the solutions produced by the exact solver Concorde [2] are used as the ground-truth solutions. Once well trained, given a new TSP instance with m vertices (randomly distributed within an unit square), the model is able to build a heat map, which estimates the probability P_{ij} of each edge (i, j) belonging to the optimal solution.

[1] https://github.com/Spider-scnu/TSP

[2] https://github.com/jvkersch/pyconcorde

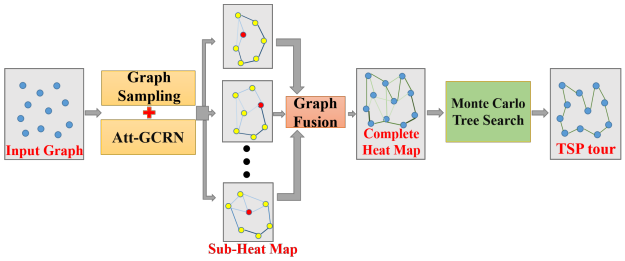


Figure 1: Pipeline of the proposed approach

Pipeline

Given a TSP instance of arbitrarily large size, the pipeline for solving this instance is shown in Fig. 1, which consists of three main steps. Respectively, the first step (off-line learning) uses a graph sampling method to extract from the original graph a number of sub-graphs (each exactly consists of m vertices), and then uses the pre-trained Att-GCRN model to build a sub heat map corresponding to each sub-graph. After that, the second step tries to merge all the sub heat maps into a complete heat map (corresponding to the original graph). Finally, the third step uses a reinforcement learning method (online learning), i.e., Monte Carlo tree search (MCTS), to search high-quality TSP solutions, guided by the information stored in the merged heat map.

Building and Merging Heat Maps

The pre-trained model is able to build a heat map of a TSP instance with m vertices. However, it can not be directly used to handle instances of different size. To deal with this issue, an optional approach is to train a series of models with different sizes, like the choice of (Joshi, Laurent, and Bresson 2019). Unfortunately, this approach seems unreasonable for very large TSP instances, since the supervised learning process requires a large number of pre-computed optimal (at least high-quality) solutions, being unaffordable for large-scale TSP instances. To avoid repetitively training models, in this paper we develop a series of techniques (illustrated in Fig. 2 and described as follows), to extend the predication ability of the fix-sized model to arbitrarily large TSP instances.

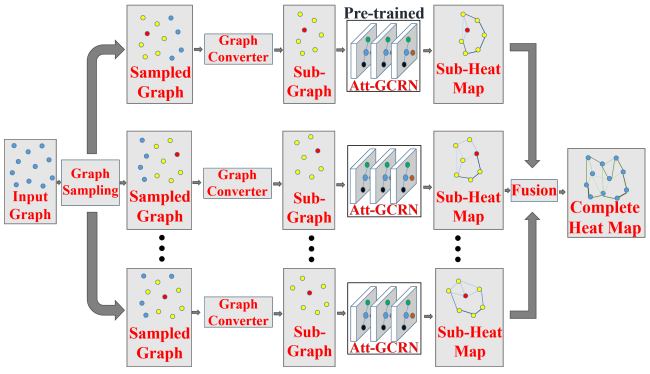


Figure 2: Method for building and merging heat maps

Graph Sampling The graph sampling method is used to extract a number of sub-graphs (each with m vertices) from the original graph G. To do this, for each vertex $i \in V$ or each edge $(i,j) \in E$, let O_i or O_{ij} (initialized to 0) respectively denote the times that vertex i (or edge (i,j)) belongs to an extracted sub-graph. Then, at each iteration, we choose the vertex i with the minimal value of O_i (randomly choose one if there are multiple such vertices) as the clustering center, and use the k-nearest neighbors algorithm (Dudani 1976) to extract a sub-graph G' which consists of exactly m vertices (including the clustering center). Then, for each vertex i or each edge (i,j) belonging to G', let $O_i \leftarrow O_i + 1$, $O_{ij} \leftarrow O_{ij} + 1$.

Above process is repeated, until the minimal value of O_i reaches a lower bound ω (a pre-defined parameter). Notice that the extracted sub-graphs may overlap, i.e., any vertex or edge may belong to different sub-graphs.

Graph Converting For each instance of the train set, all the vertices are distributed randomly within an unit square. To make sure the extracted sub-graph G' also meets this distribution, we should convert it to a new graph G''. For this purpose, let $x^{min} = \min_{i \in G'} x_i$, $x^{max} = \max_{i \in G'} x_i$, $y^{min} = \min_{i \in G'} y_i$, $y^{max} = \max_{i \in G'} y_i$ respectively denote the minimal, maximal value of the horizonal and vertical coordinates among all the m vertices of G', and let $s = \frac{1}{\max(x^{max} - x^{min}, y^{max} - y^{min})}$ be an amplification factor. Then, for each vertex $i \in G'$, we convert its coordinates (x_i, y_i) to new coordinates (x_i^{new}, y_i^{new}):

$$\begin{aligned} x_i^{new} &\leftarrow s \times (x_i - x^{min}), \\ y_i^{new} &\leftarrow s \times (y_i - y^{min}). \end{aligned} \tag{1}$$

After that, sub-graph G' is converted to a new graph G''.

Building Sub Heat Maps For each converted sub-graph G'', the coordinates of the m vertices are fed into the pre-trained Att-GCRN model, to build a sub heat map over G''.

Merging Sub Heat Maps Above two steps are repeated, thus we can obtain a number (denoted by I) of sub heat maps. Finally, we try to merge them into a complete heat map. To do this, for each edge (i,j) of the original graph G, we estimate its probability P_{ij} of belonging to the optimal TSP solution as follows.

$$P_{ij} = \frac{1}{O_{ij}} \times \sum_{l=1}^{I} P_{ij}^{''}(l). \tag{2}$$

where $P_{ij}^{''}(l)$ denotes the probability of edge (i,j) (after conversion) belonging to the optimal solution of the lth converted sub-graph G'' .

After merging all the sub heat maps, we obtain a complete heat map over the original graph G. Then, all the edges with $P_{ij} < 10^{-4}$ are marked as unpromising edges, which are eliminated directly to reduce the search space.

Reinforcement Learning for Solutions Optimization

Based on the heat map obtained above, we develop a reinforcement learning based approach to search high-quality solutions. The search process is considered as a Markov Decision Process (MDP), which starts from an initial state $\boldsymbol{\pi}$, and iteratively applies an action $\boldsymbol{a}$ to reach a new state $\boldsymbol{\pi}^*$. The details are described as follows.

States and actions In our implementation, each state corresponds to a complete TSP solution, i.e., a permutation $\boldsymbol{\pi} = (\pi_1, \pi_2, \ldots, \pi_n)$ of all the vertices. Each action $\boldsymbol{a}$ is a transformation which converts a given state $\boldsymbol{\pi}$ to a new state $\boldsymbol{\pi}^*$. Since each TSP solution consists of a subset of n edges, any action could be viewed as a k-opt ($2 \leq k \leq n$) transformation, which deletes k edges at first, and then adds k different edges to form a new tour.

Obviously, each action can be represented as a series of $2k$ sub-decisions (k edges to delete and k edges to add). This representation method is straightforward, but seems a bit redundant, since the deleted edges and added edges are highly relevant, while arbitrarily deleting k edges and adding k edges may result in an unfeasible solution. Inspired by the LK operator developed in (Lin and Kernighan 1973), we use a compact method to represent an action, which consists of only k sub-decisions. Formally, an action can be represented as $\boldsymbol{a} = (a_1, b_1, a_2, b_2, \ldots, a_k, b_k, a_{k+1})$, where k is a variable and the final vertex must coincide with the first vertex, i.e. $a_{k+1} = a_1$. Each action corresponds to a k-opt transformation, which deletes k edges, i.e., $(a_i, b_i), 1 \leq i \leq k$, and adds k edges, i.e., $(b_i, a_{i+1}), 1 \leq i \leq k$, to reach a new state. Notice that not all these elements are optional. Once a_i is known, b_i can be uniquely determined without any optional choice (explained and exemplified in the full version of this paper). Therefore, to determine an action we should only decide a series of k sub-decisions, i.e., the k vertices $a_i, 1 \leq i \leq k$. Additionally, an action involving an unpromising edge (b_i, a_{i+1}), i.e., $P_{b_i a_{i+1}} < 10^{-4}$, is marked as an unpromising action and eliminated directly.

Intuitively, this compact representation method brings advantages in two-folds: (1) fewer (only k, not $2k$) sub-decisions need to be made; (2) the resulting states are always feasible solutions (thus do not need to check feasibility).

Let $L(\boldsymbol{\pi})$ denote the tour length corresponding to state $\boldsymbol{\pi}$, then corresponding to each action $\boldsymbol{a} = (a_1, b_1, a_2, b_2, \ldots, a_k, b_k, a_{k+1})$ which converts $\boldsymbol{\pi}$ to a new state $\boldsymbol{\pi}^*$, the difference $\Delta(\boldsymbol{\pi}, \boldsymbol{\pi}^*) = L(\boldsymbol{\pi}^*) - L(\boldsymbol{\pi})$ could be calculated as follows:

$$\Delta(\boldsymbol{\pi}, \boldsymbol{\pi}^*) = \sum_{i=1}^{k} d_{b_i a_{i+1}} - \sum_{i=1}^{k} d_{a_i b_i}. \tag{3}$$

If $\Delta(\boldsymbol{\pi}, \boldsymbol{\pi}^*) < 0$, $\boldsymbol{\pi}^*$ is better (with shorter tour length) than $\boldsymbol{\pi}$.

State Initialization For state initialization, we choose a constructive procedure, which starts from an arbitrarily chosen begin vertex π_1, iteratively selects a vertex $\pi_i, 2 \leq i \leq n$ among the candidate (unvisited) vertices and adds it to the end of the partial tour, until forming a complete tour $\boldsymbol{\pi} = (\pi_1, \pi_2, \ldots, \pi_n)$. More precisely, at the ith iteration, if there are more than one candidate vertices, each candidate vertex j is chosen with a probability proportional to $\exp(P_{\pi_i j})$, while all the candidate vertices share a total probability of 1.

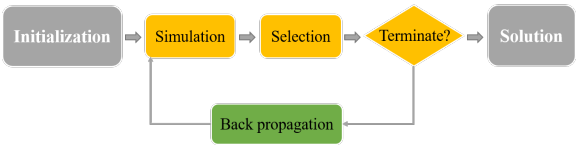


Figure 3: Procedure of the Monte Carlo tree search

Enumerating within Small Neighborhood To maintain the generalization ability of our approach, we avoid to use complex hand-crafted rules, such as the α-nearness criterion in (Helsgaun 2000) and the POPMUSIC strategy in (Taillard and Helsgaun 2019), which have proven to be highly effective on the TSP, but heavily depend on expert knowledge. Instead, starting from a new state, we at first use a straightforward method to search within a small neighborhood. More precisely, the method examines one by one the promising actions with $k = 2$, and iteratively applies the first-met improving action which leads to a better state, until no improving action with $k = 2$ is found. This simple method is able to efficiently and robustly converge to a local optimal state.

Targeted Sampling within Enlarged Neighborhood Once no improving action is found within the small neighborhood, we switch to an enlarged neighborhood which consists of the actions with $k > 2$. Unfortunately, there are generally a huge number of actions within the enlarged neighborhood (even after eliminating the unpromising ones), being impossible to enumerate them one by one. Therefore, we choose to sample a subset of promising actions (guided by RL) and iteratively select an action to apply, to reach a new state.

Following this idea, we choose the Monte Carlo tree search (MCTS) as our learning framework. Inspired by the works in (Coulom 2006), (Browne et al. 2012), (Silver et al. 2016) and (Silver et al. 2017), our MCTS procedure (outlined in Fig. 3) consists of four steps, i.e., (1) Initialization, (2) Simulation, (3) Selection, and (4) Back-propagation, which are respectively designed as follows.

Initialization: We define two $n \times n$ symmetric matrices, i.e., a weight matrix $\boldsymbol{W}$ whose element W_{ij} (initialized to $100 \times P_{ij}$) controls the probability of choosing vertex j after vertex i, and an access matrix $\boldsymbol{Q}$ whose element Q_{ij} (initialized to 0) records the times that edge (i, j) is chosen during simulations. Additionally, a variable M (initialized to 0) is used to record the total number of actions already simulated. Note that this initialization step should be executed only once at the beginning of the whole process of MDP.

Simulation: Given a state $\boldsymbol{\pi}$, we use the simulation process to probabilistically generate a number of actions. As explained before, each action is represented as $\boldsymbol{a} = (a_1, b_1, a_2, b_2, \ldots, a_k, b_k, a_{k+1})$, containing a series of sub-

decisions $a_i, 1 \leq i \leq k$ (k is also a variable, and $a_{k+1} = a_1$), while b_i could be determined uniquely once a_i is known. Once b_i is determined, for each edge $(b_i, j), j \neq b_i$, we use the following formula to estimate its potential $Z_{b_i j}$ (the higher the value of $Z_{b_i j}$, the larger the opportunity of edge (b_i, j) to be chosen):

$$Z_{b_i j} = \frac{W_{b_i j}}{\Omega_{b_i}} + \alpha \sqrt{\frac{\ln (M+1)}{Q_{b_i j} + 1}}. \tag{4}$$

Where $\Omega_{b_i} = \frac{\sum_{j \neq b_i} W_{b_i j}}{\sum_{j \neq b_i} 1}$ denotes the averaged $W_{b_i j}$ value of all the edges relative to vertex b_i. In this formula, the left part $\frac{W_{b_i j}}{\Omega_{b_i}}$ emphasizes the importance of the edges with high $W_{b_i j}$ values (to enhance the intensification feature), while the right part $\sqrt{\frac{\ln (M+1)}{Q_{b_i j} + 1}}$ prefers the rarely examined edges (to enhance the diversification feature). α is a parameter used to achieve a balance between intensification and diversification, and the term "+1" is used to avoid a negative numerator or a zero denominator.

To make the sub-decisions sequently, we at first choose a_1 randomly, and determine b_1 subsequently. Recursively, once a_i and b_i are known, a_{i+1} is decided as follows: (1) if closing the loop (connecting b_i to a_1) would lead to an improving action, or $i \geq 10$, let $a_{i+1} = a_1$. (2) otherwise, consider the vertices with $W_{b_i j} \geq 1$ as candidate vertices, forming a set $\mathbb{X}$ (excluding a_1 and the vertex already connected to b_i). Then, among $\mathbb{X}$ each vertex j is selected as a_{i+1} with probability P_j, which is determined as follows:

$$P_j = \frac{Z_{b_i j}}{\sum_{l \in \mathbb{X}} Z_{b_i l}}. \tag{5}$$

Once $a_{i+1} = a_1$, we close the loop to obtain an action.

Similarly, more actions are generated (forming a sampling pool), until meeting an improving action which leads to a better state, or the number of actions reaches its upper bound (controlled by a parameter H).

Selection: During above simulation process, if an improving action is met, it is selected and applied to the current state $\boldsymbol{\pi}$, to get a new state $\boldsymbol{\pi}^{new}$. Otherwise, if no such action exists in the sampling pool, it seems difficult to gain improvement within the current search area. Then, the MDP jumps to a random state (using the state initialization method described above), which serves as a new starting state.

Back-propagation: The value of M as well as the elements of matrices $\boldsymbol{W}$ and $\boldsymbol{Q}$ are updated by back propagation as follows. At first, whenever an action is examined, M is increased by 1. Then, for each edge (b_i, a_{i+1}) which appears in an examined action, let $Q_{b_i a_{i+1}}$ increase by 1. Finally, whenever a state $\boldsymbol{\pi}$ is converted to a better state $\boldsymbol{\pi}^{new}$ by applying action $\boldsymbol{a} = (a_1, b_1, a_2, b_2, \ldots, a_k, b_k, a_{k+1})$, for each edge $(b_i, a_{i+1}), 1 \leq i \leq k$, let:

$$W_{b_i a_{i+1}} \leftarrow W_{b_i a_{i+1}} + \beta \left[\exp\left(\frac{L(\boldsymbol{\pi}) - L(\boldsymbol{\pi}^{new})}{L(\boldsymbol{\pi})} \right) - 1 \right]. \tag{6}$$

Where β is a parameter used to control the increasing rate of $W_{b_i a_{i+1}}$. Notice that we update $W_{b_i a_{i+1}}$ only when meeting a better state, since we want to avoid wrong estimations (even in a bad action which leads to a worse state, there may exist some good edges (b_i, a_{i+1})). With this back-propagation process, the weight of the good edges would be increased to enhance its opportunity of being selected, thus the sampling process would be more and more targeted.

$\boldsymbol{W}$ and $\boldsymbol{Q}$ are symmetric matrices, thus let $W_{a_{i+1} b_i} = W_{b_i a_{i+1}}$ and $Q_{a_{i+1} b_i} = Q_{b_i a_{i+1}}$ always.

Termination Condition The MCTS iterates through the simulation, selection and back-propagation steps, until no improving action exists among the sampling pool. Then, the MDP jumps to a new state, and launches a new round of search within small and enlarged neighborhood again. This process is repeated, until the allowed time (controlled by a parameter T) has been elapsed. Then, the best found state is returned as the final solution.

Experiments

To evaluate the performance of our method, we program the algorithm for building heat maps in Python, and program the MCTS algorithm in C++ language [3]. Then, we carry out experiments on a large number of TSP instances, and make comparisons with eight newest learning based baselines, as well as three strong non-learning algorithms (the programming and training details about the baselines are given in the full version of this paper). Notice that, for the baselines, we just directly download and rerun the source codes, based on the pre-trained models (only for learning based baselines) which are publicly available. To ensure fair comparisons, all the learning based baselines as well as our new algorithm are uniformly executed on one GTX 1080 Ti GPU (to fully utilize the computing resources, as many instances as possible are executed in parallel). For the three non-learning algorithms, their source codes currently do not support running on GPU, thus we re-run them on one Intel(R) Xeon(R) Gold 5118 CPU @ 2.30GHz (with 8 cores), and list the results just for indicative purposes. Notice that our method is a learning based algorithm, thus we do not aim to strictly outperform the non-learning algorithms.

Data Sets

We use two data sets: (1) **Set 1** [4], which is divided into three subsets, each containing 10,000 automatically generated 2D-Euclidean TSP instances, respectively with $n = 20, 50, 100$. This data set is widely used by the existing learning based algorithms. (2) **Set 2**, following the same rules, we newly generate 400 larger instances, i.e., 128 instances respectively with $n = 200, 500, 1000$, and 16 instances with $n = 10000$.

Parameters

As described before, our method relies on six hyper parameters (m, ω, α, β, H and T). For parameter m which controls

[3] Publicly available at https://github.com/Spider-scnu/TSP.

[4] Downloaded from https://drive.google.com/file/d/1-5W-S5e7CKsJ9uY9uVXIyxgbcZZNYBrp/view.

the size of the pre-trained model, we set $m = 20$ for the small instances of data set 1, and set $m = 50$ for the large instances of data set 2. For the following four parameters, we uniformly choose $\omega = 5$, $\alpha = 1$, $\beta = 10$, $H = 10n$ as the default settings. Finally, for parameter T which controls the termination time, we respectively set $T = 10n$ and $T = 40n$ milliseconds for each instance of data set 1 and data set 2, to ensure that our algorithm elapses no more time than the best (in terms of solution quality) learning algorithm proposed in each reference paper.

Results on Data Set 1

Table 1 presents the results obtained by our algorithm (Att-GCRN+MCTS) on data set 1, with respect to the existing baselines. Respectively, the first three lines list two exact solvers, i.e., Concorde [5] and Gurobi [6], as well as one strong heuristic LKH3 (Helsgaun 2017). The following eight lines are all learning based algorithms which combine traditional operations for post-optimization. There are also several End-to-End ML models in the literature, but they all produce very poor results, thus being omitted here. For the columns, column 1 indicates the methods, while column 2 indicates the type of each algorithm. Columns 3-5 respectively give the average tour length, average gap in percentage w.r.t. Concorde, and the total clock time used by each algorithm on all the 10,000 instances with $n = 20$. To ensure fair comparisons, for some learning baselines, the original parameters (such as the width of beam search) are adapted to prolong the total running time. These adapted results are indicated in underlines in the table. For our method (last line), the time is divided into two parts, i.e., the time for building heat maps plus the time for running MCTS. Columns 6-8, 9-11 respectively give the same information on the instances with $n = 50$ and 100.

As shown in Table 1, the three non-learning algorithms obtain good results on all the test instances, while the existing learning based algorithms all struggle to match optimality on the instances with $n = 100$. Compared to these baselines, our algorithm performs quite well, which succeeds in matching the ground-truth solutions (reported by Concorde) on most of these instances, corresponding to an average gap of **0.0000%**, **0.0145%**, **0.0370%** respectively on instances with $n = 20, 50, 100$. The total runtime of our method remains competitive w.r.t. all the learning baselines only except two (with greedy heuristics), which are deterministic thus the results cannot be improved by prolonging the runtime.

Results on Data Set 2

At first, we summarize in Table 2 the results obtained on the 384 instances with $n = 200, 500, 1000$. Concorde and LKH3 still perform well on these instances, while Gurobi performs well on the instances with n=200 and 500, but fails to terminate within reasonable time on the instances with 1000 cities. For the learning baselines, they all produce results far away from optimality, especially on the instances with 1000 vertices. By contrast, our method is able to obtain, within short time, results very close to optimality (corresponding to a gap of **0.8844%**, **2.5365%** and **3.2238%** respectively on the instances with $n = 200$, 500 and 1000), clearly outperforming the existing learning baselines.

Furthermore, we evaluate the performance of Att-GCN+MCTS on the 16 largest instances with 10,000 vertices (see Table 3). On these large instances, several baseline algorithms face a big challenge. For example, the three learning based algorithms proposed in (Joshi, Laurent, and Bresson 2019) all fail due to memory exception (tested on the same platform as previously described), while the two exact solvers (Concorde and Gurobi) as well as the two GAT models in (Deudon et al. 2018) all fail due to time exception (up to five hours is allowed for each instance). Therefore, we exclude these seven baseline algorithms, and just compare our Att-GCRN+MCTS algorithm with the remaining three learning based algorithms (Kool, van Hoof, and Welling 2019), all evaluated on one GTX 1080 Ti GPU. The results produced by LKH3 (evaluated on one Intel(R) Xeon(R) Gold 5118 CPU @ 2.30GHz) are listed for indicative purpose. As shown in Table 3, Att-GCRN+MCTS is able to produce solutions close to LKH3, corresponding to a small average gap of **4.3902**%. By contrast, the three learning based algorithms correspond to a huge average gap of 501.2737%, 97.3932%, 80.2802% respectively. The runtime of our algorithm remains reasonable (shorter than the best one of the three baselines).

Additionally, we would like to mention three recent learning based TSP algorithms, i.e., Shimomura and Takashima (2016), Xing and Tu (2020) and Wu et al. (2020). The source codes of these three papers are all not publicly available, thus we cannot evaluate them uniformly on the same platform. In Shimomura and Takashima (2016), the authors did not report instance-per-instance results, thus it seems impossible for us to make direct comparisons with this method. In Xing and Tu (2020), on the test instances with 20, 50, 100, 200, 500, 1000 cities, the authors respectively claimed an average gap of 0.01%, 0.20%, 1.04%, 1.91%, 4.37%, 4.48% w.r.t. optimality (all worse than ours). In Wu et al. (2020), on the instances with 20, 50, 100 cities, the authors respectively claimed an average gap of 0.00%, 0.20%, 1.42% w.r.t. optimality. Roughly speaking, compared to these two recent algorithms, our algorithm is able to produce overall better results within reasonable time (although evaluated on different platforms).

Ablation Study about Heat Map

To emphasize the importance of the heat map, for each instance with $n \geq 100$, we assign an equal probability to each edge, and rerun the MCTS algorithm alone to search solutions. The results are summarized in Table 4, where the left part lists the results obtained by the original Att-GCRN+MCTS algorithm, and the right part lists the results obtained by MCTS alone (without heat map). Clearly, after disabling the heat map, the performance of the algorithm decreases drastically, corresponding to a huge gap with respect to optimality on each data set. For comparison, the original Att-GCRN+MCTS algorithm produces results very close to optimality on each data set. These comparisons clearly certificate the value of the method for identifying promising candidate edges.

[5] Downloaded from https://github.com/jvkersch/pyconcorde

[6] See https://www.gurobi.com

Method	Type	TSP20			TSP50			TSP100		
		Length	Gap	Time	Length	Gap	Time	Length	Gap	Time
Concorde	Exact Solver	3.8303	0.0000%	2.31m	5.6906	0.0000%	13.68m	7.7609	0.0000%	1.04h
Gurobi	Exact Solver	3.8302	-0.0001%	2.33m	5.6905	0.0000%	26.20m	7.7609	0.0000%	3.57h
LKH3	Heuristic	3.8303	0.0000%	20.96m	5.6906	0.0013%	26.65m	7.7611	0.0026%	49.96m
GAT (Deudon et al. 2018)	RL, S	3.8741	1.1443%	10.30m	6.1085	7.3438%	19.52m	8.8372	13.8679%	47.78m
GAT (Deudon et al. 2018)	RL, S, 2OPT	3.8501	0.5178%	15.62m	5.8941	3.5759%	27.81m	8.2449	6.2365%	4.95h
GAT (Kool et al. 2018)	RL, S	3.8322	0.0501%	16.47m	5.7185	0.4912%	22.85m	7.9735	2.7391%	1.23h
GAT (Kool et al. 2018)	RL, G	3.8413	0.2867%	6.03s	5.7849	1.6568%	34.92s	8.1008	4.3791%	1.83m
GAT (Kool et al. 2018)	RL, BS	3.8304	0.0022%	15.01m	5.7070	0.2892%	25.58m	7.9536	2.4829%	1.68h
GCN (Joshi et al. 2019)	SL, G	3.8552	0.6509%	19.41s	5.8932	3.5608%	2.00m	8.4128	8.3995%	11.08m
GCN (Joshi et al. 2019)	SL, BS	3.8347	0.1158%	21.35m	5.7071	0.2905%	35.13m	7.8763	1.4828%	31.80m
GCN (Joshi et al. 2019)	SL, BS*	3.8305	0.0075%	22.18m	5.6920	0.02509%	37.56m	7.8719	1.4299%	1.20h
Att-GCRN+MCTS(**Ours**)	SL+RL	3.8303	**0.0000%**	23.33s + 1.25m	5.6914	**0.0145%**	2.59m + 5.33m	7.7638	**0.0370%**	3.94m + 10.62m

Table 1: Our results w.r.t. the baselines, tested on 10,000 instances respectively with n=20, 50 and 100 (some results of Concorde are not strictly optimal, possibly due to the reason of integer approximation, so as the results in Table 2).

Method	Type	TSP200			TSP500			TSP1000		
		Length	Gap	Time	Length	Gap	Time	Length	Gap	Time
Concorde	Solver	10.7191	0.0000%	3.44m	16.5458	0.0000%	37.66m	23.1182	0.0000%	6.65h
Gurobi	Solver	10.7036	-0.1446%	40.49m	16.5171	-0.1733%	45.63h	-	-	-
LKH3	Heuristic	10.7195	0.0040%	2.01m	16.5463	0.0029%	11.41m	23.1190	0.0036%	38.09m
GAT (Deudon et al. 2018)	RL, S	13.1746	22.9079%	4.84m	28.6291	73.0293%	20.18m	50.3018	117.5860%	37.07m
GAT (Deudon et al. 2018)	RL, S, 2OPT	11.6104	8.3159%	9.59m	23.7546	43.5687%	57.76m	47.7291	106.4575%	5.39h
GAT (Kool et al. 2018)	RL, S	11.4497	6.8160%	4.49m	22.6409	36.8382%	15.64m	42.8036	85.1519%	63.97m
GAT (Kool et al. 2018)	RL, G	11.6096	8.3081%	5.03s	20.0188	20.9902%	1.51m	31.1526	34.7539%	3.18m
GAT (Kool et al. 2018)	RL, BS	11.3769	6.1364%	5.77m	19.5283	18.0257%	21.99m	29.9048	29.2359%	1.64h
GCN (Joshi et al. 2019)	SL, G	17.0141	58.7272%	59.11s	29.7173	79.6063%	6.67m	48.6151	110.2900%	28.52m
GCN (Joshi et al. 2019)	SL, BS	16.1878	51.0185%	4.63m	30.3702	83.5523%	38.02m	51.2593	121.7278%	51.67m
GCN (Joshi et al. 2019)	SL, BS*	16.2081	51.2079%	3.97m	30.4258	83.8883%	30.62m	51.0992	121.0357%	3.23h
Att-GCN+MCTS (**Ours**)	SL+RL	10.8139	**0.8844%**	20.62s + 2.15m	16.9655	**2.5365%**	31.17s + 5.39m	23.8634	**3.2238%**	43.94s + 11.74m

Table 2: Our results w.r.t. existing baselines, tested on 128 instances respectively with n=200, 500 and 1000.

Method	TSP10000		
	Length	Gap (vs. LKH3)	Time
LKH3	71.7778	-	8.8h
GAT (Kool et al. 2018)	431.5812	501.2737%	12.63m
GAT (Kool et al. 2018)	141.6846	97.3932%	5.99m
GAT (Kool et al. 2018)	129.4012	80.2802%	1.81h
Att-GCN+MCTS (**Ours**)	74.9290	**4.3902%**	4.16m + 1.69h

Table 3: Performance of Att-GCRN+MCTS w.r.t. four baselines, tested on 16 TSP instances with 10,000 vertices.

Instance	Att-GCRN+MCTS			MCTS (without heat map)		
	Length	Opt. Gap.	Time	Length	Opt. Gap.	Time
TSP100	7.76	0.04%	3.94m+ 10.62m	46.43	498.31%	10.37m
TSP200	10.81	0.88%	20.63s+ 2.15m	96.50	800.25%	2.10m
TSP500	16.97	2.54%	31.73s+ 5.39m	247.88	1398.15%	4.81m
TSP1000	23.86	3.22%	43.94s+ 11.74m	502.51	2073.67%	10.43m
TSP10000	74.93	4.39%	4.16m+ 1.69h	1000.02	1293.22%	1.56h

Table 4: Ablation study about the heat map.

Conclusions

Supervised learning based methods require a large amount of training data, being difficult to generalize to large-scale TSP instances. This research shows that, it is possible to train a small-scale model in supervised manner, and smoothly generalize it to tackle large TSP instances, by applying a series of techniques such as graph sampling, graph converting and heat maps merging. This method can inherit the advantages of supervised learning, and avoid repetitively training models of different sizes. Experimental results confirmed that, this method is able to develop highly competitive learning based TSP algorithm, and significantly improve the generalization ability of the pre-trained model. In the future, we will try to solve extremely large or non-Euclidean TSP instances.

Acknowledgements

We would like to thank the anonymous reviewers for their insightful comments that helped to considerably improve the paper. This paper was supported in part by the Shenzhen Science and Technology Innovation Commission under grant JCYJ20180508162601910, the National Key R&D Program of China under grant 2020YFB1313300, and the Funding from the Shenzhen Institute of Artificial Intelligence and Robotics for Society under grant 2019-INT003. Jia-Ming Xin also contributed to this paper.

References

Applegate, D. L.; Bixby, R. E.; Chvátal, V.; Cook, W.; Espinoza, D. G.; Goycoolea, M.; and Helsgaun, K. 2009. Certification of an optimal TSP tour through 85,900 cities. *Operations Research Letters* 37(1): 11–15.

Bello, I.; Pham, H.; Le, Q. V.; Norouzi, M.; and Bengio, S. 2017. Neural combinatorial optimization with reinforcement learning. In *Proceeding of the International Conference on Learning Representations (ICLR)*.

Bengio, Y.; Lodi, A.; and Prouvost, A. 2018. Machine Learning for Combinatorial Optimization: a Methodological Tour d'Horizon. *arXiv preprint arXiv:1811.06128* .

Browne, C. B.; Powley, E.; Whitehouse, D.; Lucas, S. M.; Cowling, P. I.; Rohlfshagen, P.; Tavener, S.; Perez, D.; Samothrakis, S.; and Colton, S. 2012. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in games* 4(1): 1–43.

Chen, X.; and Tian, Y. 2019. Learning to perform local rewriting for combinatorial optimization. In *Advances in Neural Information Processing Systems*, 6278–6289.

Coulom, R. 2006. Efficient selectivity and backup operators in Monte-Carlo tree search. In *International conference on computers and games*, 72–83. Springer.

Deudon, M.; Cournut, P.; Lacoste, A.; Adulyasak, Y.; and Rousseau, L.-M. 2018. Learning heuristics for the tsp by policy gradient. In *International Conference on the Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, 170–181. Springer.

Dudani, S. A. 1976. The distance-weighted k-nearest-neighbor rule. *IEEE Transactions on Systems, Man, and Cybernetics* (4): 325–327.

Emami, P.; and Ranka, S. 2018. Learning permutations with sinkhorn policy gradient. *arXiv preprint arXiv:1805.07010* .

Guo, T.; Han, C.; Tang, S.; and Ding, M. 2019. Solving Combinatorial Problems with Machine Learning Methods. In *Nonlinear Combinatorial Optimization*, 207–229. Springer.

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

Helsgaun, K. 2017. An extension of the Lin-Kernighan-Helsgaun TSP solver for constrained traveling salesman and vehicle routing problems. *Roskilde: Roskilde University* .

Hopfield, J. J.; and Tank, D. W. 1985. Neural computation of decisions in optimization problems. *Biological cybernetics* 52(3): 141–152.

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* .

Kaempfer, Y.; and Wolf, L. 2019. Learning the multiple traveling salesmen problem with permutation invariant pooling networks. In *Proceeding of the International Conference on Learning Representations (ICLR)*.

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 (NeurIPS)*, 6348–6358.

Kool, W.; van Hoof, H.; and Welling, M. 2019. Attention, Learn to Solve Routing Problems! In *International Conference on Learning Representations (ICLR)*.

Lin, S.; and Kernighan, B. W. 1973. An effective heuristic algorithm for the traveling-salesman problem. *Operations Research* 21(2): 498–516.

Lu, H.; Zhang, X.; and Yang, S. 2020. A Learning-based Iterative Method for Solving Vehicle Routing Problems. In *International Conference on Learning Representations (ICLR)*.

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 (NeurIPS)*, 9839–9849.

Nowak, A.; Villar, S.; Bandeira, A. S.; and Bruna, J. 2017. A note on learning algorithms for quadratic assignment with graph neural networks. In *Proceeding of the 34^{th} International Conference on Machine Learning (ICML)*, volume 1050, 22.

Prates, M.; Avelar, P. H.; Lemos, H.; Lamb, L. C.; and Vardi, M. Y. 2019. Learning to Solve NP-Complete Problems: A Graph Neural Network for Decision TSP. In *Proceedings of the AAAI Conference on Artificial Intelligence (AAAI)*, volume 33, 4731–4738.

Rego, C.; Gamboa, D.; Glover, F.; and Osterman, C. 2011. Traveling salesman problem heuristics: Leading methods, implementations and latest advances. *European Journal of Operational Research* 211(3): 427–441.

Shimomura, M.; and Takashima, Y. 2016. Application of Monte-Carlo Tree Search to Traveling-Salesman Problem. In *The 20th Workshop on Synthesis And System Integration of Mixed Information technologies (SASIMI)*, 352–356.

Silver, D.; Huang, A.; Maddison, C. J.; Guez, A.; Sifre, L.; Van Den Driessche, G.; Schrittwieser, J.; Antonoglou, I.; Panneershelvam, V.; Lanctot, M.; et al. 2016. Mastering the game of Go with deep neural networks and tree search. *Nature* 529(7587): 484.

Silver, D.; Schrittwieser, J.; Simonyan, K.; Antonoglou, I.; Huang, A.; Guez, A.; Hubert, T.; Baker, L.; Lai, M.; Bolton, A.; et al. 2017. Mastering the game of go without human knowledge. *Nature* 550(7676): 354.

Taillard, É. D.; and Helsgaun, K. 2019. POPMUSIC for the travelling salesman problem. *European Journal of Operational Research* 272(2): 420–429.

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

Wu, Y.; Song, W.; Cao, Z.; Zhang, J.; and Lim, A. 2020. Learning Improvement Heuristics for Solving Routing Problems. *arXiv preprint arXiv:1912.05784* .

Xing, Z.; and Tu, S. 2020. A Graph Neural Network Assisted Monte Carlo Tree Search Approach to Traveling Salesman Problem. In *IEEE Access*. doi:10.1109/ACCESS.2020.3000236.

Appendix of AAAI2021 paper: Generalize a Small Pre-trained Model to Arbitrarily Large TSP Instances

1. Architecture of the Att-GCRN model

As shown in Fig. 1, the Att-GCRN model uses three steps to convert the input information (the coordinates of each vertex, as well as the length of each edge) into a heat map, i.e., the probability of each edge belonging to the optimal solution. At first, it embeds the input information into a series of vectors (each vertex or each edge corresponds to a 300×1 vector). Then, a feature extractor is used to extract the weights information of the vertices and edges. Finally, the information is fed into a classification network to build the heat map.

Additionally, to train the model, the solutions produced by exact solver Concorde are used as the ground-truth solutions, while a focal loss function is chosen as the loss function, and the adaptive moment estimation method (Kingma & Ba, 2014) is used as the gradient descent optimizer (with an initial learning rate of 10^{-2}).

1.1. Graph embedding

Given an undirected graph G with n vertices, we at first embed each vertex and each edge into a vector (Levy & Goldberg, 2014) as follows:

$$\begin{aligned} \boldsymbol{v}_i^0 &= \boldsymbol{W}_1 \boldsymbol{c}_i + \boldsymbol{A}_i, \\ \boldsymbol{e}_{ij}^0 &= \boldsymbol{W}_2 \boldsymbol{d}_{ij} + \boldsymbol{B}_{ij}. \end{aligned} \tag{1}$$

where $\boldsymbol{v}_i^0$ and $\boldsymbol{e}_{ij}^0$ respectively denotes the embedded vector (300×1) corresponding to vertex i and edge (i, j). Respectively, $\boldsymbol{W}_1$ is a 300×2 matrix, $\boldsymbol{c}_i$ denotes the coordinates of vertex i (represented as a 2×1 vector), $\boldsymbol{A}_i$ is a 300×1 vector corresponding to vertex i, $\boldsymbol{W}_2$ is a 300×1 vector, $\boldsymbol{d}_{ij}$ denotes the distance of edge (i, j), $\boldsymbol{B}_{ij}$ is also a 300×1 vector corresponding to edge (i, j). Notice that all the elements of $\boldsymbol{W}_1$, $\boldsymbol{W}_2$, $\boldsymbol{A}_i$ and $\boldsymbol{B}_{ij}$ are trainable parameters, which are randomly initialized, and updated via gradient descent optimizer during the training process.

Overall, the embedding of all the vertices forms a $300 \times n$ matrix $\boldsymbol{v}^0$, and the embedding of all the edges forms a $300 \times n \times n$ tensor $\boldsymbol{e}^0$, which are fed into the feature extractor.

1.2. Feature extractor

The feature extractor consists of L (L=30 in this paper) graph residual convolutional layers. Let $\mathbf{H}^l = \{\boldsymbol{v}^l, \boldsymbol{e}^l\}$ denote the graph feature of the lth layer, whose elements are recursively calculated as follows:

$$\begin{aligned} \boldsymbol{v}_i^l = \boldsymbol{v}_i^{l-1} + \mathrm{ReLU}\Big(&\mathrm{BN}\big(\mathrm{Conv1d}(\boldsymbol{v}_i^{l-1}) \\ &+ \sum_{j \neq i} \alpha_{ij}^l \odot \mathrm{Conv1d}(\boldsymbol{v}_j^{l-1})\big)\Big), \\ \boldsymbol{e}_{ij}^l = \boldsymbol{e}_{ij}^{l-1} + \mathrm{ReLU}\Big(&\mathrm{BN}\big(\mathrm{Conv1d}^*(\boldsymbol{e}_{ij}^{l-1}) \\ &+ \mathrm{Conv1d}(\boldsymbol{v}_i^{l-1}) + \mathrm{Conv1d}(\boldsymbol{v}_j^{l-1})\big)\Big). \end{aligned} \tag{2}$$

where ReLU denotes rectified linear unit, BN denotes batch normalization, $\mathrm{Conv1d}$ denotes a one-dimensional convolutional layer with the stride of 1 and 300 kernels of size 1×1 (Glorot et al., 2011; Ioffe & Szegedy, 2015). $\mathrm{Conv1d}^*$ is similar to $\mathrm{Conv1d}$, but with 300 kernels of size $1 \times 1 \times 1$ (the output of $\mathrm{Conv1d}(\boldsymbol{v}_i^{l-1})$ and $\mathrm{Conv1d}^*(\boldsymbol{e}_{ij}^{l-1})$ are both 300×1 vectors). α_{ij}^l denotes the attention coefficient relative to edge (i, j), which is defined as follows:

$$\alpha_{ij}^l = \frac{\exp\Big(\mathrm{LeakyReLU}\big(\mathrm{Conv1d}^\diamond(\boldsymbol{e}_{ij}^l)\big)\Big)}{\sum_{k \neq i} \exp\Big(\mathrm{LeakyReLU}\big(\mathrm{Conv1d}^\diamond(\boldsymbol{e}_{ik}^l)\big)\Big)}, \tag{3}$$

where $\mathrm{LeakyReLU}$ denotes leaky rectified linear unit, $\mathrm{Conv1d}^\diamond$ is a one-dimensional convolutional layer which only consists of one $1 \times 1 \times 1$ kernel with the stride of 1 (Maas et al., 2013), and $\mathrm{Conv1d}^\diamond(\boldsymbol{e}_{ij}^l)$ outputs a real value.

1.3. Classification network

Based on the information of $\boldsymbol{e}^L$, the classification network tries to build a heat map by feed-forward propagation:

$$\boldsymbol{P} = \mathrm{MLP}(\boldsymbol{e}^L) \tag{4}$$

where MLP denotes a multi-layer perception (Cybenko, 1989) with three full connected layers. $\boldsymbol{P}$ denotes the heat map (a $n \times n$ matrix), whose element $\boldsymbol{P}_{ij}$ estimates the probability of edge (i, j) belonging to the optimal solution.

1.4. Loss function

The model in trained in batch (each batch contains M=500 training instances), and a focal loss function (Lin et al.,

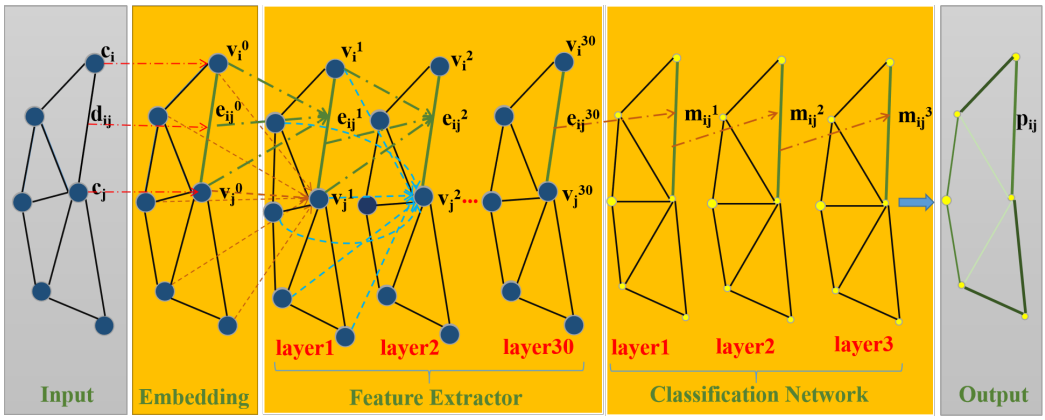


Figure 1. Architecture of the Att-GCRN model.

2017) is used to calculate the accumulated loss:

$$\mathrm{FL}(\boldsymbol{P}) = -\frac{1}{M}\sum_{m=1}^{M}\sum_{i=1}^{N}\sum_{j\neq i}^{N}\Big(\beta_0(1-\boldsymbol{I}_{ij}^{(m)}) + \beta_1 \times \mathbf{I}_{ij}^{(m)}\Big)\Big(\mathbf{I}_{ij}^{(m)} - \boldsymbol{P}_{ij}^{(m)}\Big)\log(\boldsymbol{P}_{ij}^{(m)}). \tag{5}$$

$\boldsymbol{I}^{(m)}$ is a $n \times n$ matrix corresponding the mth training instance, whose element $\boldsymbol{I}_{ij}^{(m)} = 1$ if edge (i, j) belongs to the ground-truth solution of the m-th training instance, and $\boldsymbol{I}_{ij}^{(m)} = 0$ otherwise. $\boldsymbol{P}_{ij}^{(m)}$ is an element of the m-th heat map (built by the classification network). β_0 and β_1 are balanced coefficients relative to the edges not belonging to (or belonging to) the ground-truth solutions, which are calculated as follows (Liu & Zhou, 2006):

$$\beta_0 = \frac{n^2}{n^2 - 2n},\ \beta_1 = \frac{n^2}{2n}. \tag{6}$$

1.5. Additional information

Training platform: one GTX 1080 Ti GPU.

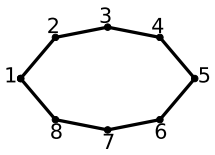
Training set: we train two models, i.e., TSP20-model and TSP50-model (respectively, each input graph has 20 or 50 vertices), which are respectively used to tackle small-scale (with $n \leq 100$) or large-scale (with $n > 100$) instances. Respectively, we use the 990,000 instances with 20 vertices (50 vertices) generated in (Joshi et al., 2019) to train our TSP20-model (TSP50-model).

Training time: 12 hours for TSP20-model (970 epoches), 25 hours for TSP50-model (1270 epoches), excluding the time elapsed by Concorde to produce ground-truth solutions.

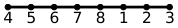
2. Example of applying an action during MDP

As described previously, during the Markov decision process (MDP), each action is represented as $\boldsymbol{a} = (a_1, b_1, a_2, b_2, \ldots, a_k, b_k, a_{k+1})$, where k is a variable and the final vertex a_{k+1} must coincide with a_1.

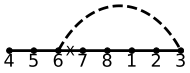
Fig.2 exemplifies the process of applying an action. Respectively, sub-figure (a) is the starting state $\boldsymbol{\pi} = (1, 2, 3, 4, 5, 6, 7, 8)$. To determine an action, we at first decide a vertex a_1 and delete the edge between a_1 and its previous vertex b_1. Without loss of generality, suppose $a_1 = 4$, then $b_1 = 3$ and edge $(3, 4)$ is deleted, resulting in a temporary status shown in sub-figure (b) (for the sake of clarity, drawn as a line which starts from a_1 and ends at b_1). Furthermore, we decide a vertex a_2 to connect vertex b_1, generally resulting in an unfeasible solution containing an inner cycle (unless $a_2 = a_1$). For example, suppose $a_2 = 6$ and connect it to vertex 3, the resulting temporary status is shown in sub-figure (c), where an inner cycle occurs and the degree of vertex a_2 increases to 3. To break inner cycle and reduce the degree of a_2 to 2, the edge between vertex a_2 and vertex $b_2 = 7$ should be deleted, resulting in a temporary status shown in sub-figure (d). This process is repeated, to get a series of vertices a_k and b_k ($k \geq 2$). In this example, $a_3 = 3$ and $b_3 = 2$, respectively corresponding to sub-figures (e) and (f). Once $a_{k+1} = a_1$, the loop closes and reaches a new state (feasible TSP solution). For example, if let $a_4 = a_1 = 4$ and connect a_4 to b_3, the resulting new state is shown in sub-figure (g), which is redrawn as a cycle in sub-figure (h).



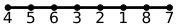
(a) starting state



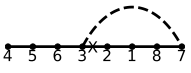
(b) after deleting (3,4)



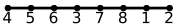
(c) after connecting 6 to 3



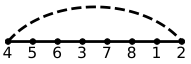
(d) after deleting (7,6)



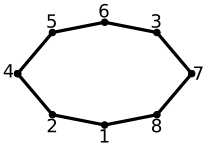
(e) after connecting 3 to 7



(f) after deleting (2,3)



(g) after connecting 4 to 2



(h) new state

Figure 2. The decision process of an example action

3. Programming and training information about the baselines and our algorithm

Detailed in Table 1.

Reference

Cybenko, G. Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems*, 2(4):303–314, 1989.

Deudon, M., Cournut, P., Lacoste, A., Adulyasak, Y., and Rousseau, L.-M. 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.

Glorot, X., Bordes, A., and Bengio, Y. Deep sparse rectifier neural networks. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, pp. 315–323, 2011.

Ioffe, S. and Szegedy, C. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

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

Kingma, D. P. and Ba, J. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

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

Levy, O. and Goldberg, Y. Neural word embedding as implicit matrix factorization. In *Advances in neural information processing systems*, pp. 2177–2185, 2014.

Lin, T.-Y., Goyal, P., Girshick, R., He, K., and Dollár, P. Focal loss for dense object detection. In *Proceedings of the IEEE international conference on computer vision*, pp. 2980–2988, 2017.

Liu, X.-Y. and Zhou, Z.-H. The influence of class imbalance on cost-sensitive learning: An empirical study. In *Sixth International Conference on Data Mining (ICDM'06)*, pp. 970–974. IEEE, 2006.

Maas, A. L., Hannun, A. Y., and Ng, A. Y. Rectifier nonlinearities improve neural network acoustic models. In *Proc. icml*, volume 30, pp. 3, 2013.

Table 1. Information about the baselines as well as our Att-GCN+MCTS algorithm.

Method	Type	Programming Language	Training Time			Training Platform		
			n=20	n=50	n=100	n=20	n=50	n=100
Concorde Gurobi LKH3	Exact Solver Exact Solver Heuristic	Python Python C++	-	-	-	-	-	-
GAT (Deudon et al., 2018) GAT (Deudon et al., 2018)	RL, S RL, S, 2OPT	Python Python	$\approx$ 2h	$\approx$ 2h	$\approx$ 2h	Two GPUs Tesla K80	Two GPUs Tesla K80	Two GPUs Tesla K80
GAT (Kool et al., 2019) GAT (Kool et al., 2019) GAT (Kool et al., 2019)	RL, S RL, G RL, BS	Python Python Python	9.17h	27.22h	45.83h	Signle GPU 1080 Ti	Signle GPU 1080 Ti	Two GPUs 1080 Ti
GCN (Joshi et al., 2019) GCN (Joshi et al., 2019) GCN (Joshi et al., 2019)	SL, G SL, BS SL, BS*	Python Python Python	N/A	N/A	N/A	Single GPU 1080 Ti	Single GPU 1080 Ti	Four GPUs 1080 Ti
Att-GCN+MCTS (**Ours**)	SL+RL	Python & C++	$\approx$ 12h	$\approx$25h	-	Single GPU 1080 Ti	Single GPU 1080 Ti	-

SL:Supervised learning **RL**:Reinforcement learning **G**:Greedy **S**:Sampling **BS**:Beam search **BS***:BS and shortest tour **2OPT**:2-opt search