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

Anonymous Authors[1]

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 show that, this new approach clearly outperforms the existing machine learning based TSP algorithms, and significantly improves the generalization ability of the trained model.

1. Introduction

The travelling salesman problem (TSP) is a well-known combinatorial optimization problem with various real-life applications, such as transportation, 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 & 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.

[1]Anonymous Institution, Anonymous City, Anonymous Region, Anonymous Country. Correspondence to: Anonymous Author <anon.email@domain.com>.

Preliminary work. Under review by the International Conference on Machine Learning (ICML). Do not distribute.

- **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 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 et al. (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 & Takashima, 2016) and (Xing & 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 1,000 cities (an order of magnitude larger than the instances commonly 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.

2. 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 & Helsgaun, 2019) for an overlook of the leading TSP algorithms.

The idea of applying ML to solve the TSP is not new, dated back to several decades ago. Hopfield & Tank (1985) proposed a Hopfield-network, which achieved the best TSP solutions at that time. Encouraged by this progress, neural networks were subsequently applied on many related problems (surveyed by Smith (1999)). However, these early attempts only achieved limited success, possibly due to the lack of high-performance hardware and big data. In recent years, benefited from the rapidly improving hardware and exponentially increasing data, ML (especially deep learning) achieved great successes in the field of artificial intelligence. Motivated by these successes, ML becomes again a hot and promising topic for combinatorial optimization. A number of ML based algorithms have been developed for the TSP, which can be classified into two categories.

Supervised learning (SL) methods: Vinyals et al. (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 et al. (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 & 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 & 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) and Kool et al. (2019) both proposed a graph attention network (GAN), which incorporates attention mechanism with RL to auto-regressively improve the quality of the obtained solution.

In addition to the works focused on the classic TSP, 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 & Wolf, 2019), and the vehicle routing problem (Nazari et al., 2018), (Chen & Tian, 2019) and (Lu et al., 2020), etc. For an overall survey, please refer to (Bengio et al., 2018) and (Guo et al., 2019).

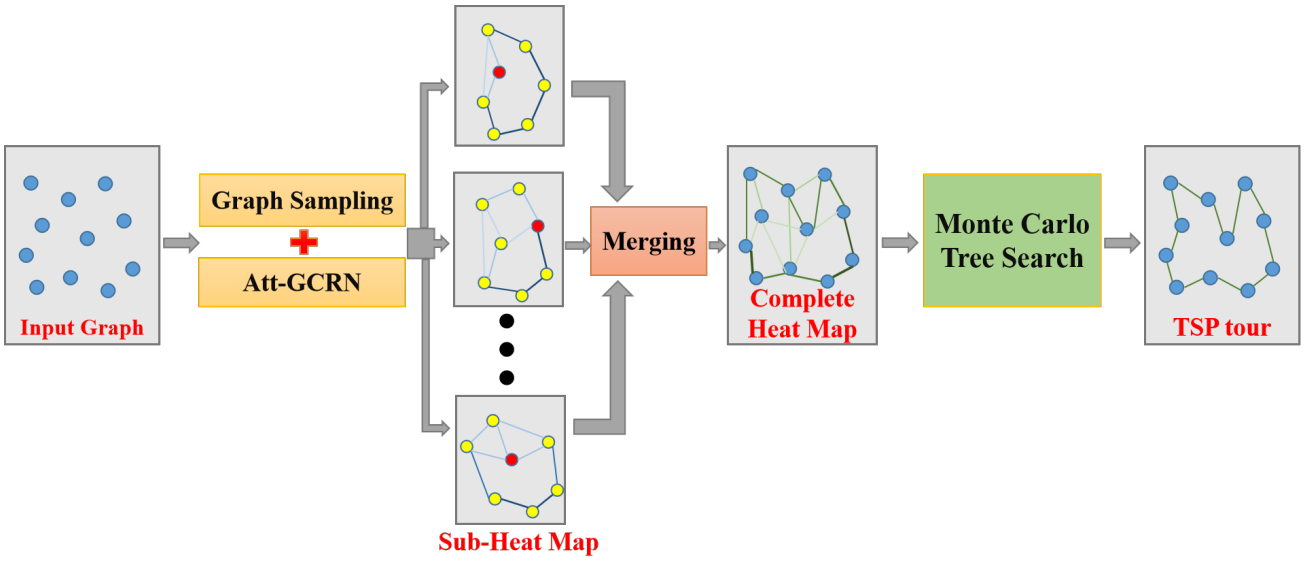

Figure 1. Pipeline of the proposed approach

3. Methods

3.1. 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 supplementary materials), 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 (Applegate et al., 2006) 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.

3.2. 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.

3.3. 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 et al., 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.

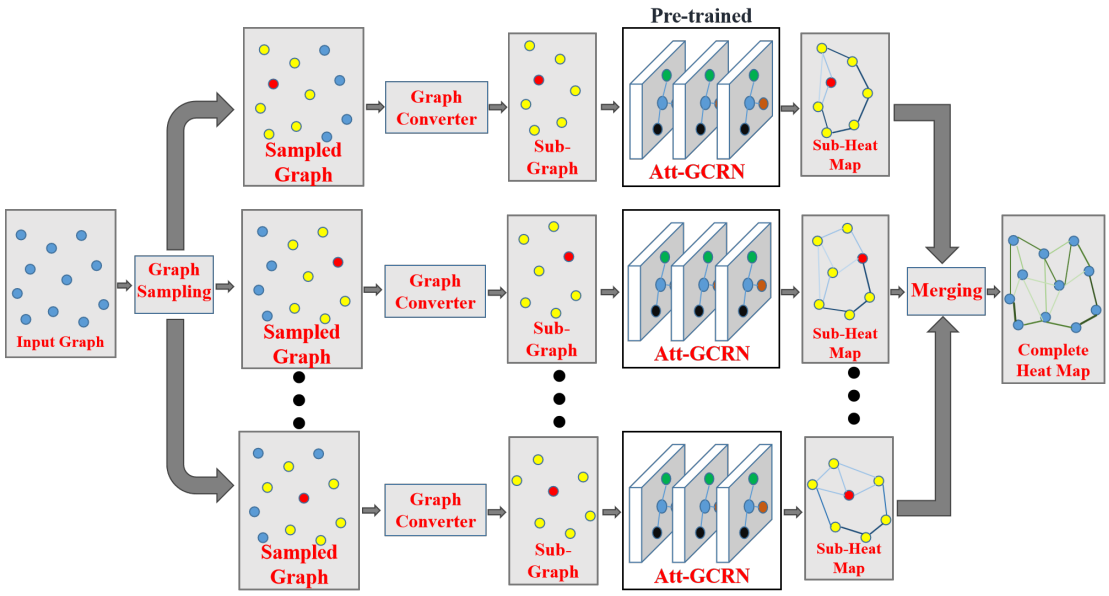

Figure 2. Method for building and merging heat maps

3.3.1. 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.

3.3.2. 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, the sub-graph G' is converted to a new graph G''.

3.3.3. 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''.

3.3.4. 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.

3.4. 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.

3.4.1. 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. To overcome this drawback, we develop 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 supplementary materials). 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 necessarily feasible solutions.

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 $\boldsymbol{\pi}$.

3.4.2. 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.

3.4.3. 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 & 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.

3.4.4. 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.

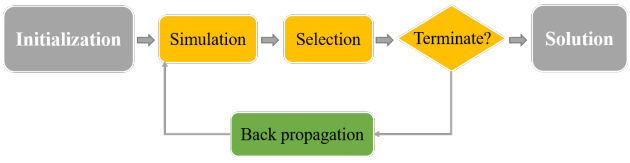

Figure 3. Procedure of the Monte Carlo tree search

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 in Section 3.4.1, 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 minus 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 method described in Section 3.4.2), 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.

3.4.5. 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.

Table 1. Information about the baselines as well as our algorithm.

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

4. 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 [1]. 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 (briefly described in Table 1, where the first three lines list non-learning algorithms). 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.

4.1. Data sets

We use two data sets: (1) **Set 1** [2], 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**[3], following the same rules, we newly generate 384 larger instances, i.e., 128 instances respectively with $n = 200, 500, 1000$.

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

[2]See https://drive.google.com/file/d/1-5W-S5e7CKsJ9uY9uVXIyxgbcZZNYBrp/view.

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

4.2. Parameters

As described in Section 3, our method relies on six hyper parameters (m, ω, α, β, H and T). For parameter m which controls 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. A sensitivity analysis about the parameters is given in the supplementary materials.

4.3. Results on data set 1

Table 2 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 (Applegate et al., 2006) [4] and Gurobi [5], 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

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

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

Table 2. Results of Att-GCRN+MCTS with respect to existing baselines, tested on 10,000 instances respectively with n=20, 50 and 100.

Method	Type	TSP20			TSP50			TSP100		
		Length	Gap	Time	Length	Gap	Time	Length	Gap	Time
Concorde	Exact Solver	3.8305	0.0000%	2.31m	5.6918	0.0000%	13.68m	7.7645	0.0000%	1.04h
Gurobi	Exact Solver	3.8302	-0.0075%	2.33m	5.6905	-0.0211%	26.20m	7.7609	-0.0456%	3.57h
LKH3	Heuristic	3.8303	-0.0074%	20.96m	5.6906	-0.0198%	26.65m	7.7611	-0.0430%	49.96m
GAT (Deudon et al., 2018)	RL, S	3.8741	1.1367%	10.30m	6.1085	7.3211%	19.52m	8.8372	13.8160%	47.78m
GAT (Deudon et al., 2018)	RL, S, 2OPT	3.8501	0.5102%	15.62m	5.8941	3.5540%	27.81m	8.2449	6.1881%	4.95h
GAT (Kool et al., 2019)	RL, S	3.8322	0.0426%	16.47m	5.7185	0.4699%	22.85m	7.9735	2.6922%	1.23h
GAT (Kool et al., 2019)	RL, G	3.8413	0.2792%	6.03s	5.7849	1.6353%	34.92s	8.1008	4.3315%	1.83m
GAT (Kool et al., 2019)	RL, BS	3.8304	-0.0053%	15.01m	5.7070	0.2680%	25.58m	7.9536	2.4361%	1.68h
GCN (Joshi et al., 2019)	SL, G	3.8552	0.6433%	19.41s	5.8932	3.5389%	2.00m	8.4128	8.3500%	11.08m
GCN (Joshi et al., 2019)	SL, BS	3.8347	0.1082%	21.35m	5.7071	0.2692%	35.13m	7.8763	1.4365%	31.80m
GCN (Joshi et al., 2019)	SL, BS*	3.8305	0.0000%	22.18m	5.6920	0.0039%	37.56m	7.8719	1.3837%	1.20h
Att-GCRN+MCTS(**Ours**)	SL+RL	3.8303	**-0.0075%**	23.33s + 2.35m	5.6914	**-0.0066%**	2.59m + 5.70m	7.7638	**-0.0086%**	3.94m + 11.34m

of beam search) are adapted to prolong the total running time. These adapted results are indicated in blue color 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 2, 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 or improving the ground-truth solutions (reported by Concorde) on most of these instances, corresponding to an average gap of **-0.0075%**, **-0.0066%**, **-0.0086%** respectively on the instances with $n = 20, 50, 100$ (on many instances, the solutions produced by Concorde are not strictly optimal, as explained in the supplementary materials). The total runtime of our method remains competitive with respect to all the learning baselines only except two (with greedy heuristics), which are deterministic thus the results cannot be improved by prolonging the runtime.

4.4. Results on data set 2

Furthermore, we summarize in Table 3 the results obtained on the 384 large instances of data set 2. 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 0.7976%, 2.3037% and 2.7857% respectively on the instances with $n = 200$, 500 and 1000), clearly outperforming the existing learning baselines. These results confirm that our method could be smoothly generalized to large instances.

Notice that although our method performs slightly worse than Concorde and LKH3 on these instances, however, as a learning based method which tries to automate the search process without complicated expert knowledge, our method is highly flexible and could be easily adapted to tackle other challenging combinatorial optimization problems. Actually, as detailed in the supplementary materials, our method outperforms Concorde on even larger instances (with 10,000 cities), where Concorde fails to terminate within reasonable time. Furthermore, we have extended the idea to tackle the capacitated vehicle routing problem (CVRP), and develop a learning based algorithm which outperforms LKH3 on many CVRP instances (detailed in another paper), showing its generalization ability among different problems.

Additionally, we would like to mention two MCTS based TSP algorithms, i.e., (Shimomura & Takashima, 2016) and Xing & Tu (2020), while the later is still a working paper. The source codes of these two papers are both not publicly available, thus we cannot evaluate them uniformly on the same platform to make strictly fair comparisons. In (Shimomura & Takashima, 2016), the authors did not report detailed results. Instead, they only provided some rough statistics (drawn in figures) to analyze the performances of different variants of their algorithm. Therefore, it seems impossible for us to make direct comparisons with this method. In Xing & Tu (2020), on the test instances with 20, 50, 100, 200, 500 cities, the authors respectively claimed an average gap of 0.03%, 0.32%, 1.53%, 1.91%, 4.37% with respect to optimality (all worse than ours), while the time elapsed on each instance was much longer than ours. Roughly speaking, compared to this recent MCTS algorithm, our algorithm is able to produce overall better results within reasonable time (although evaluated on different platforms).

Table 3. Results of Att-GCRN+MCTS with respect to existing baselines, tested on 128 instances respectively with n=200, 500 and 1000.

Method	Type	TSP200			TSP500			TSP1000		
		Length	Gap	Time	Length	Gap	Time	Length	Gap	Time
Concorde	Solver	10.7280	0.0000%	3.44m	16.5836	0.0000%	37.66m	23.2268	0.0000%	6.65h
Gurobi	Solver	10.7036	-0.2278%	40.49m	16.5171	-0.4005%	45.63h	-	-	-
LKH3	Heuristic	10.7195	-0.0793%	2.01m	16.5463	-0.2248%	11.41m	23.1190	-0.4641%	38.09m
GAT (Deudon et al., 2018)	RL, S	13.1746	22.8055%	4.84m	28.6291	72.6355%	20.18m	50.3018	116.5683%	37.07m
GAT (Deudon et al., 2018)	RL, S, 2OPT	11.6104	8.2256%	9.59m	23.7546	43.2419%	57.76m	47.7291	105.4918%	5.39h
GAT (Kool et al., 2019)	RL, S	11.4497	6.7270%	4.49m	22.6409	36.5267%	15.64m	42.8036	84.2859%	63.97m
GAT (Kool et al., 2019)	RL, G	11.6096	8.2178%	5.03s	20.0188	20.7147%	1.51m	31.1526	34.1236%	3.18m
GAT (Kool et al., 2019)	RL, BS	11.3769	6.0479%	5.77m	19.5283	17.7570%	21.99m	29.9048	28.8368%	1.64h
GCN (Joshi et al., 2019)	SL, G	17.0141	58.5949%	59.11s	29.7173	78.1988%	6.67m	48.6151	109.3064%	28.52m
GCN (Joshi et al., 2019)	SL, BS	16.1878	50.8926%	4.63m	30.3702	83.1344%	38.02m	51.2593	119.8947%	51.67m
GCN (Joshi et al., 2019)	SL, BS*	16.2081	51.0819%	3.97m	30.4258	83.4697%	30.62m	51.0992	120.0017%	3.23h
Att-GCN+MCTS (**Ours**)	SL+RL	10.8136	**0.7976%**	20.62s + 4.29m	16.9656	**2.3037%**	31.17s + 11.07m	23.8738	**2.7857%**	43.94s + 22.38m

4.5. Ablation study about heat map

To emphasize the importance of the heat map, for each instance, 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.

Table 4. Ablation study about the heat map.

Instance	Att-GCRN+MCTS			MCTS (without heat map)		
	Length	Opt. Gap.	Time	Length	Opt. Gap.	Time
TSP20	3.8303	-0.0075%	23.33s+ 2.35m	6.9712	81.9893%	2.33m
TSP50	5.6914	-0.0066%	2.59m+ 5.70m	21.0231	269.3593%	5.39m
TSP100	7.7638	-0.0086%	3.94m+ 11.34m	45.1700	481.7529%	10.68m
TSP200	10.8133	0.7948%	20.63s+ 2.15m	94.7429	783.1346%	2.13m
TSP500	16.9658	2.3049%	31.73s+ 5.36m	248.1310	1396.2471%	5.24m
TSP1000	23.8719	2.7775%	43.94s+ 10.79m	502.1278	2061.8503%	10.32m

Furthermore, for each subset of test instances, we statistic in Table 5 three indicators, i.e., (1) **ANPE**: the Average Number of Promising Edges relative to each vertex. (2)**MR**: we rank the edges relative to each vertex (in descending order according to the probability), and calculate the Mean Rank of the edges belonging to the ground-truth solutions (produced by Concorde). (3) **ANMGE**: the Average Number of Missed Good Edges, which belong to the ground-truth solutions, but are eliminated while building the heat map.

Table 5. Statistical results

Data set	ANPE	MR	ANMGE
TSP20	3.13	1.62	0.0001
TSP50	6.97	2.04	0.6268
TSP100	8.27	1.92	0.2034
TSP200	8.65	2.13	0.6512
TSP500	8.98	2.69	2.3178
TSP1000	9.15	3.87	6.2868

As shown in the table, a small number of edges are identified as promising edges (see ANPE) relative to each vertex, while very few good edges (see ANMGE) belonging to the ground-truth solutions are missed. The MR values are fairly low (the lower, the better). These results show that, our method for building heat maps is able to accurately identify promising edges while eliminating the unpromising ones, thus significantly reducing the search space.

5. Conclusions

Supervised learning based techniques are useful for discovering common patterns, but 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 methohd can inherit the advantages of supervised learning, and avoid repetitively training models of different sizes. Experimental results confirmed that, following this approach, it 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 larger TSP instances or non-Euclidean TSP instances, and extend the method to other challenging combinatorial optimization problems.

References

Applegate, D., Bixby, R., Chvatal, V., and Cook, W. Concorde tsp solver. http://www.math.uwaterloo.ca/tsp/concorde, 2006.

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

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

Bengio, Y., Lodi, A., and Prouvost, A. Machine learning for combinatorial optimization: a methodological tour d'horizon. *arXiv preprint arXiv:1811.06128*, 2018.

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

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

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

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.

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

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

Guo, T., Han, C., Tang, S., and Ding, M. Solving combinatorial problems with machine learning methods. In *Nonlinear Combinatorial Optimization*, pp. 207–229. Springer, 2019.

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

Helsgaun, K. An extension of the lin-kernighan-helsgaun tsp solver for constrained traveling salesman and vehicle routing problems. *Roskilde: Roskilde University*, 2017.

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

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.

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

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

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

Lu, H., Zhang, X., and Yang, S. A learning-based iterative method for solving vehicle routing problems. In *International Conference on Learning Representations (ICLR)*, 2020.

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

Nowak, A., Villar, S., Bandeira, A. S., and Bruna, J. 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, pp. 22, 2017.

Prates, M., Avelar, P. H., Lemos, H., Lamb, L. C., and Vardi, M. Y. 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, pp. 4731–4738, 2019.

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

Shimomura, M. and Takashima, Y. Application of monte-carlo tree search to traveling-salesman problem. In *The 20th Workshop on Synthesis And System Integration of Mixed Information technologies (SASIMI)*, pp. 352–356, 2016.

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. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484, 2016.

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

Smith, K. A. Neural networks for combinatorial optimization: a review of more than a decade of research. *INFORMS Journal on Computing*, 11(1):15–34, 1999.

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

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

Xing, Z. and Tu, S. A graph neural network assisted monte carlo tree search approach to traveling salesman problem. In *the submission of International Conference on Learning Representations (ICLR)*, 2020. URL https://openreview.net/forum?id=Syg6fxrKDB.