Combining Reinforcement Learning with Lin-Kernighan-Helsgaun Algorithm for the Traveling Salesman Problem

Jiongzhi Zheng [1,2], **Kun He**[1*], **Jianrong Zhou**[1], **Yan Jin**[1], **Chu-Min Li** [1,3]
[1]School of Computer Science, Huazhong University of Science and Technology, China
[2]Institute of Artificial Intelligence, Huazhong University of Science and Technology, China
[3]MIS, University of Picardie Jules Verne, France
brooklet60@hust.edu.cn

Abstract

We address the Traveling Salesman Problem (TSP), a famous NP-hard combinatorial optimization problem. And we propose a variable strategy reinforced approach, denoted as VSR-LKH, which combines three reinforcement learning methods (Q-learning, Sarsa and Monte Carlo) with the well-known TSP algorithm, called Lin-Kernighan-Helsgaun (LKH). VSR-LKH replaces the inflexible traversal operation in LKH, and lets the program learn to make choice at each search step by reinforcement learning. Experimental results on 111 TSP benchmarks from the TSPLIB with up to 85,900 cities demonstrate the excellent performance of the proposed method.

Introduction

Given a set of cities with certain locations, the Traveling Salesman Problem (TSP) is to find the shortest route, along which a salesman travels from a city to visit all the cities exactly once and finally returns to the starting point. An algorithm designed for TSP can also be applied to many other practical problems, such as the Vehicle Routing Problem (VRP) (Nazari et al. 2018), tool path optimization of Computer Numerical Control (CNC) machining (Fok et al. 2019), etc. As one of the most famous NP-hard combinatorial optimization problems, TSP has become a touchstone for the algorithm design.

Numerous approaches have been proposed for solving the TSP. Traditional methods are mainly exact algorithms and heuristic algorithms, such as the exact solver Corconde [1] and the Lin-Kernighan-Helsgaun (LKH) heuristic (Helsgaun 2000; Helsgaun 2009; Taillard and Helsgaun 2019; Tinós, Helsgaun, and Whitley 2018). With the development of artificial intelligence, there are some studies combining reinforcement learning (Sutton and Barto 1998) with (meta) heuristics to solve the TSP (de O. da Costa et al. 2020; Liu and Zeng 2009), and attaining good results on instances with up to 2,400 cities. More recently, Deep Reinforcement Learning (DRL) methods (Khalil et al. 2017; Shoma, Daisuke, and Hiroyuki 2018; Xing, Tu, and Xu 2020; de O. da Costa et al. 2020) have also been used to solve the TSP. However, DRL methods are hard to scale to large instances with thousands of cities, indicating that current DRL methods still have a gap to the competitive heuristic algorithms.

Heuristic algorithms are currently the most efficient and effective approaches for solving the TSP, including real-world TSPs with millions of cities. The LKH algorithm (Helsgaun 2000), which improves the Lin-Kernighan (LK) heuristic (Lin and Kernighan 1973), is one of the most famous heuristics. LKH improves the route through the k-opt heuristic optimization method (Lin 1965), which replaces at most k edges of the current tour at each search step. The most critical part of LKH is to make choice at each search step. In the k-opt process, it has to select edges to be removed and to be added. Differs to its predecessor LK heuristic of which each city has its own candidate set recording five (default value) nearest cities, LKH used an α-value defined based on the minimum spanning tree (Held and Karp 1970) as a metric in selecting and sorting cities in the candidate set. The effect of candidate sets greatly improves the iteration speed and the search speed of both LK and LKH.

However, when selecting edges to be added in the k-opt process, LKH traverses the candidate set of the current city in ascending order of the α-value until the constraint conditions are met, which is inflexible and may limit its potential to find the optimal solution. In this work, we address the challenge of improving the LKH, and introduce a creative and distinctive method to combine reinforcement learning with the LKH heuristic. The proposed algorithm could learn to choose the appropriate edges to be added in the k-opt process by means of a reinforcement learning strategy.

Concretely, we first use three reinforcement learning methods, namely Q-learning, Sarsa and Monte Carlo (Sutton and Barto 1998), to replace the inflexible traversal operation of LKH. The performance of the reinforced LKH algorithm by any one of the three methods has been greatly improved. Besides, we found that the three methods are complementary in reinforcing the LKH. For example, some TSP instances can be solved well by Q-learning, but not by Sarsa, and vice versa. Therefore, we propose a variable strategy reinforced approach, called VSR-LKH, that combines the above three algorithms to further improve the performance. The idea of variable strategy is inspired from Variable Neighborhood Search (VNS) (Mladenovic and Hansen 1997), which lever-

*Corresponding author.

[1]http://www.math.uwaterloo.ca/tsp/concorde/index.html

ages the complementarity of different local search neighborhoods. The principal contributions are as follows:

- We propose a reinforcement learning based heuristic algorithm called VSR-LKH that significantly promotes the well-known LKH algorithm, and demonstrate its promising performance on 111 public TSP benchmarks with up to 85,900 cities.
- We define a Q-value to replace the α-value by combining the city distance and α-value for the selection and sorting of candidate cities. And Q-value can be adjusted adaptively by learning from the information of many feasible solutions generated during the iterative search.
- Since algorithms solving an NP-hard combinatorial optimization problem usually need to make choice among many candidates at each search step, our approach suggests a way to improve conventional algorithms by letting them learn to make good choices intelligently.

Related Work

This section reviews related work in solving the TSP using exact algorithms, heuristic algorithms and reinforcement learning methods respectively.

Exact Algorithms. The branch and bound (BnB) method is often used to exactly solve the TSP and its variants (Pekny and Miller 1990; Pesant et al. 1998). The best-known exact solver Corconde [1] is based on BnB, whose initial tour is obtained by the Chained LK algorithm (Applegate, Cook, and Rohe 2003). Recently, the Corconde solver was sped up by improving the initial solution using a partition crossover (Sanches, Whitley, and Tinós 2017). All of these exact algorithms can yield optimal solutions, but the computational costs rise sharply along with the problem scale.

Heuristic Algorithms. Although heuristic algorithms cannot guarantee the optimal solution, they can obtain a sub-optimal solution within reasonable time. Heuristic algorithms for solving the TSP can be divided into three categories: tour construction algorithms, tour improvement algorithms and composite algorithms.

Each step of the tour construction algorithms determines the next city the salesman will visit until the complete TSP tour is obtained. The nearest neighbor algorithm (Hougardy and Wilde 2015) and ant colony algorithm (Gao 2020) are among the most common methods for tour construction.

The tour improvement algorithms usually make improvements on a randomly initialized tour. The k-opt algorithm (Lin 1965) and genetic algorithm fall into this category. The evolutionary algorithm represented by Nagata (2006) is one of the most successful genetic algorithms to solve TSP, which is based on an improved edge assembly crossover (EAX) operation. Its selection model can maintain the population diversity at low cost.

The composite algorithms usually use tour improvement algorithms to improve the initial solution obtained by a tour construction algorithm. The famous LKH algorithm is a composite algorithm that uses k-opt to improve the heuristically constructed initial tour.

Reinforcement Learning based Methods. With the rapid development of artificial intelligence, some researchers have adopted reinforcement learning technique for solving the TSP.

One category is to combine reinforcement learning or DRL with existing (meta) heuristics. Ant-Q (Gambardella and Dorigo 1995) and Q-ACS (Sun, Tatsumi, and Zhao 2001) replaced the pheromone in the ant colony algorithm with the Q-table in the Q-learning algorithm. However, the effect of Q-table is similar to that of the pheromone. Liu and Zeng (2009) used reinforcement learning to construct mutation individuals in the successful genetic algorithm EAX-GA (Nagata 2006) and reported results on instances with up to 2,400 cities, but their proposed algorithm RMGA is inefficient compared with EAX-GA and LKH. Costa *et al.* (2020) utilized DRL algorithm to learn a 2-opt based heuristic, Wu *et al.* (2019) combined DRL method with the tour improvement approach such as 2-opt and node swap. They reported results on real-world TSP instances with up to 300 cities.

Another category is to apply DRL to directly solve the TSP. Bello *et al.* (2017) addressed TSP by using the actor-critic method to train a pointer network (Vinyals, Fortunato, and Jaitly 2015). The S2V-DQN (Khalil et al. 2017) used reinforcement learning to train graph neural networks so as to solve several combinatorial optimization problems, including minimum vertex cover, maximum cut and TSP. Shoma *et al.* (2018) used reinforcement learning with a convolutional neural network to solve the TSP. The graph convolutional network technique (Joshi, Laurent, and Bresson 2019) was also applied to solve the TSP. The ECO-DQN (Barrett et al. 2020) is an improved version of S2V-DQN, which has obtained better results than S2V-DQN on the maximum cut problem. Xing *et al.* (2020) used deep neural network combined with Monte Carlo tree search to solve the TSP. It is generally hard for them to scale to large TSP instances with thousands cities as an effective heuristic such as LKH does.

In this work, we aim to combine reinforcement learning with existing excellent heuristic algorithm in a more effective way, and handle efficiently large scale instances. We reinforce the key component of LKH, the k-opt, and thus promote the performance significantly. To our knowledge, this is the first work that combines reinforcement learning with the key search process of a well-known algorithm to solve the famous NP-hard TSP.

The Existing LKH Algorithm

We give a brief introduction to the LKH algorithm (see more details of LKH and its predecessor LK heuristic in Appendix [2]). LKH uses k-opt (Lin 1965) as the optimization method to improve the TSP tour. During the k-opt process, LKH first selects a starting point p_1, then alternately selects k edges $\{x_1, x_2, ..., x_k\}$ to be removed in the current tour and k edges $\{y_1, y_2, ..., y_k\}$ to be added until the stopping criterion is met ($2 \leq k \leq 5$, which is not predetermined).

Figure 1 shows the iterative process of k-opt in LKH. The key point is to select $2k$ cities $\{p_1, p_2, ..., p_{2k}\}$ such that for each i ($1 \leq i \leq k$), $x_i = (p_{2i-1}, p_{2i})$, for each i

[2] https://arxiv.org/abs/2012.04461

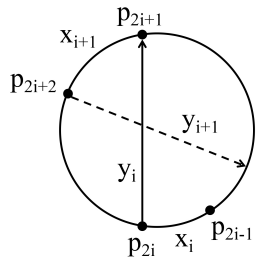


Figure 1: The choice of x_i, y_i, x_{i+1} and y_{i+1} in k-opt.

$(1 \leq i \leq k-1)$, $y_i = (p_{2i}, p_{2i+1})$ and $y_k = (p_{2k}, p_1)$. And the following conditions should be satisfied: (1) x_i and y_i must share an endpoint, and so do y_i and x_{i+1}; (2) For $i \geq 2$, if p_{2i} connects back to p_1 for y_i, the resulting configuration should be a tour; (3) y_i is always chosen so that $\sum_{j=1}^{i}(l(x_j) - l(y_j)) > 0$, where $l(\cdot)$ is the length of the corresponding edge; (4) Set $\{x_1, x_2, ..., x_i\}$ and set $\{y_1, y_2, ..., y_i\}$ are disjoint. To this end, LKH first randomly selects p_1. After p_{2i-1} is selected, p_{2i} is randomly selected from the two neighbors of p_{2i-1} in the current TSP tour. Then p_{2i+1} is selected by traversing the candidate set of p_{2i}.

The current iteration of k-opt process will quit when a k-opt move is found that can improve the current TSP tour (by trying to connect p_{2k} with p_1 as y_k before selecting p_{2k+1}) or no edge pair, x_i and y_i, satisfies the constraints.

The α-value. The candidate set of each city in LKH stores five other cities in ascending order of the α-values. To explain α-value, we need to introduce the structure of 1-tree (Held and Karp 1970). A 1-tree for a graph $G(V, E)$ (V is the set of nodes, E is the set of edges) is a spanning tree on the node set $V\backslash\{v\}$ combined with two edges from E incident to node v, which is a special point chosen arbitrarily. A *minimum* 1-tree is a 1-tree with minimum length.Obviously, the length of the *minimum* 1-tree is the lower bound of the optimal TSP solution. Suppose $L(T)$ is the length of the *minimum* 1-tree of graph $G(V, E)$ and $L(T^+(i, j))$ is the length of the *minimum* 1-tree required to contain edge (i, j), the α-value of edge (i, j) can be calculated by Eq. 1:

$$\alpha(i, j) = L(T^+(i, j)) - L(T). \tag{1}$$

Penalties. LKH uses a method to maximize the lower bound of the optimal TSP solution by adding *penalties* (Held and Karp 1970). Concretely, a π-value computed using a sub-gradient optimization method (Held and Karp 1971) is added to each node as a penalty when calculating the distance between two nodes:

$$C(i, j) = d(i, j) + \pi_i + \pi_j, \tag{2}$$

where $C(i, j)$ is the cost for a salesman from city i to city j after adding *penalties*, $d(i, j)$ is the distance between the two cities, and π_i, π_j are the *penalties* added to the two cities respectively. The *penalties* actually change the cost matrix of the TSP. Note that this change does not change the optimal solution of the TSP, but it changes the *minimum* 1-tree. Suppose $L(T_\pi)$ is the length of the *minimum* 1-tree after adding the *penalties*, then the lower bound $w(\pi)$ of the optimal solution can be calculated by Eq. 3, which is a function of set $\pi = [\pi_1, ..., \pi_n]$:

$$w(\pi) = L(T_\pi) - 2\sum_i \pi_i. \tag{3}$$

The lower bound $w(\pi)$ of the optimal solution is maximized, and after adding the *penalties*, the α-value is further improved for the candidate set.

The Proposed VSR-LKH Algorithm

The proposed Variable Strategy Reinforced LKH (VSR-LKH) algorithm combines reinforcement learning with LKH. In VSR-LKH, we change the method of traversing the candidate set in the k-opt process and let the program automatically select appropriate edges to be added in the candidate set by reinforcement learning. We further use a variable strategy method that combines the advantages of three reinforcement learning methods, so that VSR-LKH can improve the flexibility as well as the robustness and avoid falling into local optimal solutions. We achieve VSR-LKH on top of LKH, so VSR-LKH still retains the characteristics of LKH such as candidate set, *penalties* and other improvements made by Helsgaun (Helsgaun 2009; Taillard and Helsgaun 2019; Tinós, Helsgaun, and Whitley 2018).

Reinforcement Learning Framework

Since VSR-LKH lets the program learn to make correct decisions for choosing the edges to be added, the states and actions in our reinforcement learning framework are all related to the edges to be added. And an episode corresponds to a k-opt process. We use the value iteration method to estimate the *state-action function* $q_\pi(s, a) = \sum_{t=0}^{\infty}\gamma^t r(s_t, a_t)$, where (s_t, a_t) is the state-action pair at time step t of an episode, $s_0 = s, a_0 = a$, and r is the corresponding reward. The detailed description of the states, actions and rewards in the reinforcement learning framework are as follows:

- *States*: The current state of the system is a city that is going to select an edge to be added. For example in Figure 1, for edge y_i, the state corresponds to its start point p_{2i}. When $i = 1$, the initial state s_0 corresponds to $p_{2i} = p_2$.
- *Actions*: For a state s, the action is to choose another endpoint of the edge to be added except s from the candidate set of s. For example in Figure 1, for edge y_i, the action corresponds to its endpoint p_{2i+1}. When $i = 1$, action a_0 corresponds to $p_{2i+1} = p_3$.
- *Transition*: The next state after performing the action is the next city that needs to select an edge to be added. For example, p_{2i+2} is the state transferred to after executing action p_{2i+1} at state p_{2i}.
- *Rewards*: The reward function should be able to represent the improvement of the tour when taking an action at the current state. The reward $r(s_t, a_t)$ obtained by performing action a_t at state s_t can be calculated by:

$$r(s_t, a_t) = \begin{cases} C(a_{t-1}, s_t) - C(s_t, a_t) & t > 0 \\ C(p_1, s_0) - C(s_0, a_0) & t = 0 \end{cases}, \tag{4}$$

where function $C(\cdot, \cdot)$ is shown in Eq. 2.

Initial Estimation of the State-action Function

We associate the estimated value of the *state-action function* in VSR-LKH to each city and call it Q-value of the city. It is used as the basis for selecting and sorting the candidate cities. The initial Q-value $Q(i,j)$ for the candidate city j of city i is defined as follows:

$$Q(i,j) = \frac{w(\pi)}{\alpha(i,j) + d(i,j)}. \tag{5}$$

The initial Q-value defined in Eq. 5 combines the factors of the selection and sorting of candidate cities in LK (Lin and Kernighan 1973) and LKH (Helsgaun 2000), which are the distance and the α-value, respectively. Note that α-value is based on a minimum 1-tree and is rather a global property, while the distance between two cities is a local property. Combining α-value and distance can take the advantage of both properties. Although the influence of the distance factor is small, it can avoid the denominator to be 0. The purpose of $w(\pi)$ is two-folds. First, it can prevent the initial Q-value from being much smaller than the rewards. Second, it can adaptively adjust the initial Q-value for different instances. The experimental results also demonstrate that the performance of LKH can be improved by only replacing α-value with the initial Q-value defined in Eq. 5 to select and sort the candidate cities.

The Reinforced Algorithms

VSR-LKH applies reinforcement technique to learn to adjust Q-value so as to estimate the *state-action function* more accurately. As the maximum value of k in the k-opt process of LKH is as small as 5, we choose the Monte Carlo method and one-step Temporal-Difference (TD) algorithms including Q-learning and Sarsa (Sutton and Barto 1998) to improve the k-opt process in LKH.

Monte Carlo. Monte Carlo method is well-known for the model-free reinforcement learning based on averaging sample returns. In the reinforcement learning framework, there was no repetitive state or action in one or even multiple episodes. Therefore, for any state action pair (s_t, a_t), the Monte Carlo method uses the episode return after taking action a_t at state s_t as the estimation of its Q-value. That is,

$$Q(s_t, a_t) = \sum\nolimits_{i=0}^{+\infty} r(s_{t+i}, a_{t+i}). \tag{6}$$

One-step TD. TD learning is a combination of Monte Carlo and Dynamic Programming. The TD algorithms can update the Q-values in an on-line, fully incremental fashion. In this work, we use both of the on-policy TD control (Sarsa) and off-policy TD control (Q-learning) to reinforce the LKH. The one-step Sarsa and Q-learning update the Q-value respectively as follows:

$$\begin{aligned} Q(s_t, a_t) =& (1-\lambda) \cdot Q(s_t, a_t) + \\ & \lambda \cdot [r(s_t, a_t) + \gamma Q(s_{t+1}, a_{t+1})], \end{aligned} \tag{7}$$

$$\begin{aligned} Q(s_t, a_t) =& (1-\lambda) \cdot Q(s_t, a_t) + \\ & \lambda \cdot [r(s_t, a_t) + \gamma \max_{a'} Q(s_{t+1}, a')], \end{aligned} \tag{8}$$

where $\lambda \in (0,1)$ in Eq. 7 and Eq. 8 is the learning rate.

Variable Strategy Reinforced k-opt

The VSR-LKH algorithm uses the reinforcement learning technique to estimate the *state-action function* to determine the optimal policy. The policy guides the program to select the appropriate edges to be added in the k-opt process. Moreover, a variable strategy method is added to combine the advantages of the three reinforcement learning algorithms, Q-learning, Sarsa and Monte Carlo, and leverages their complementarity. When VSR-LKH judges that the current reinforcement learning method (Q-learning, Sarsa or Monte Carlo) may not be able to continue optimizing the current TSP tour, the variable strategy mechanism will switch to another method. Algorithm 1 shows the flow of the variable strategy reinforced k-opt process of VSR-LKH. And the source code of VSR-LKH is available at https://github.com/JHL-HUST/VSR-LKH/.

In Algorithm 1, we apply two functions *ChooseInitialTour*() and *CreateCandidateSet*() in LKH to initialize the initial tour of TSP and the candidate sets of all cities. VSR-LKH uses the ϵ-greedy method (Sutton and Barto 1998; Wunder, Littman, and Babes 2010) to trade-off the exploration-exploitation dilemma in the reinforcement learning process. And the value of ϵ is reduced by the attenuation coefficient β to make the algorithm inclined to exploitation as the number of iterations increases. The reinforcement learning strategy is switched if the algorithm could not improve the solution after a certain number of iterations, denoted as *MaxNum*. Each iteration in the k-opt process of VSR-LKH needs to find cities p_{2k+1} and p_{2k+2} that meet the constraints before updating the Q-value. The final TSP route is stored in *BestTour*.

Experimental Results

Experimental results provide insight on why and how the proposed approach is effective, suggesting that the performance of VSR-LKH is due to the flexibility and robustness of variable strategy reinforced learning and the benefit of our Q-value definition that combines city distance and α-value.

Experimental Setup

The experiments were performed on a personal computer with Intel® i5-4590 3.30 GHz 4-core CPU and 8 GB RAM. The parameters related to reinforcement learning in VSR-LKH are set as follows: $\epsilon = 0.4$, $\beta = 0.99$, $\lambda = 0.1$, $\gamma = 0.9$, the maximum iterations, *MaxTrials* is equal to the number of cities in the TSP, and $MaxNum = MaxTrials/20$. Actually, VSR-LKH is not very sensitive to these parameters. The performance of VSR-LKH with different parameters is compared in Appendix. Other parameters are consistent with the example given in the LKH open source website [3], and the LKH baseline used in our experiments is also from this website.

All the TSP instances are from the TSPLIB [4], including 111 symmetric TSPs. Among them, there are 77 instances with less than 1,000 cities and 34 instances with at least

[3] http://akira.ruc.dk/%7Ekeld/research/LKH/

[4] http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/

Algorithm 1 Variable strategy reinforced k-opt process

Input: ϵ-greedy parameter: ϵ, attenuation coefficient: β, *BestTour*=*ChooseInitialTour*(), maximum number of iterations: *MaxTrials*, variable strategy parameters: *MaxNum*
Output: *BestTour*

```
Initialize Q-values according to Eq. 5
CreateCandidateSet()
Initialize M = 1 that corresponds to the policy (1: Q-learning, 2: Sarsa, 3: Monte Carlo), initialize num = 0
for i = 1 : MaxTrials do
  num = num + 1, ε = ε × β
  if num ≥ MaxNum then
    M = M%3 + 1, num = 0
  end if
  BetterTour=ChooseInitialTour()
  repeat
    Randomly select an unselected edge (p1, p2) in BetterTour as the initial edge
    Initialize the set of edges to be removed R_e = ∅, the set of edges to be added A_e = ∅
    R_e := R_e ∪ {(p1, p2)}, set k = 1
    Suppose C_2k is the candidate set of p_2k
    while the stopping criterion is not satisfied do
      p_2k+1 = ε-greedy(C_2k), C_2k := C_2k \ {p_2k+1}
      if p_2k+1 does not satisfy the constraints then
        continue
      end if
      Randomly traverse the two neighbors of p_2k+1 to select p_2k+2 that satisfies the constraints
      Update Q(p_2k, p_2k+1) according to Eq. 6, 7 or 8 corresponding to M
      A_e := A_e ∪ {(p_2k, p_2k+1)}
      R_e := R_e ∪ {(p_2k+1, p_2k+2)}
      k = k + 1
    end while
    A_e := A_e ∪ {(p_2k, p1)}
    if the sum of lengths of all edges in A_e is less than that in R_e then
      Replace edges in R_e with edges in A_e and set the new tour as BetterTour
      break
    end if
  until each edge in BetterTour has been selected as the initial edge (Note that edges (i, j) and (j, i) are different)
  if length of BetterTour is less than BestTour then
    Replace BestTour with BetterTour, num = 0
  end if
  Exit if BestTour is equal to the optimum (if known)
end for
```

1,000 cities. The number of cities in these instances ranges from 14 to 85,900. Each instance is solved 10 times by each tested algorithm. An instance is considered to be *easy* if both LKH and VSR-LKH can reach the optimal solution in each of the 10 runs (for each run, the maximum number of iterations *MaxTrials* equals to the number of cities), and otherwise it is considered to be *hard*. All *hard* instances with less than 1,000 cities include: kroB150, si175, rat195, gr229, pr299, gr431, d493, att532, si535, rat575 and gr666, a total of 11. All instances with at least 1,000 cities in TSPLIB except dsj1000, si1032, d1291, u1432, d1655, u2319, pr2392 and pla7397 are *hard*, a total of 26. So there are 37 *hard* instances and 74 *easy* instances in TSPLIB. Note that the number in an instance name indicates the number of cities in that instance.

We tested VSR-LKH on all the 111 symmetric TSP instances from the TSPLIB, but mainly use results on the *hard* instances to compare different algorithms for clarity.

Q-value Evaluation

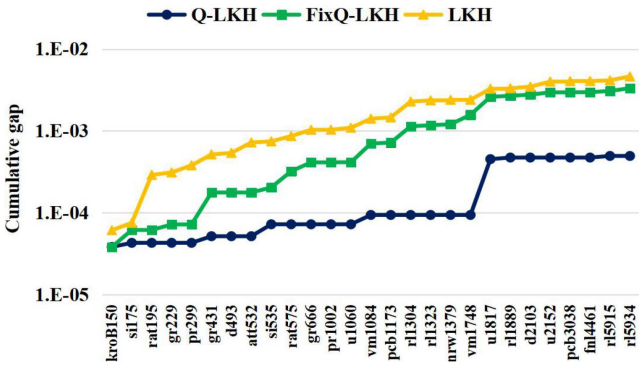
In order to demonstrate the positive influence of the Q-value we proposed in Eq. 5 on the performance, we pick 27 *hard* instances having the shortest runtime by LKH, and compare the results of three algorithms, including LKH, Q-LKH (a variant of VSR-LKH reinforced only by Q-learning, i.e. M is always 1 in Algorithm 1) and FixQ-LKH (a variant of VSR-LKH with fixed Q-value defined by Eq. 5 and without reinforcement learning). FixQ-LKH is equivalent to LKH but uses the initial Q-value defined by Eq. 5 instead of α-value to select and sort the candidate cities.

Figure 2 shows the comparison results on TSP instances. Each instance is solved 10 times by the above three algorithms. The results are expressed by the cumulative gap on solution quality and cumulative runtime, because cumulants are more intuitive than comparing the results of each instance individually. $gap(a, j) = \frac{1}{10}\sum_{i=1}^{10}\frac{A_i - Opt_j}{Opt_j}$ is the average gap of calculating the j-th instance by algorithm a, where A_i is the result of the i-th calculation and Opt_j is the optimal solution of the j-th instance. The smaller the average gap, the closer the average solution is to the optimal solution. $C_{gap}(a, j) = \sum_{i=1}^{j} gap(a, i)$ is the cumulative gap. The cumulative runtime is calculated analogously.

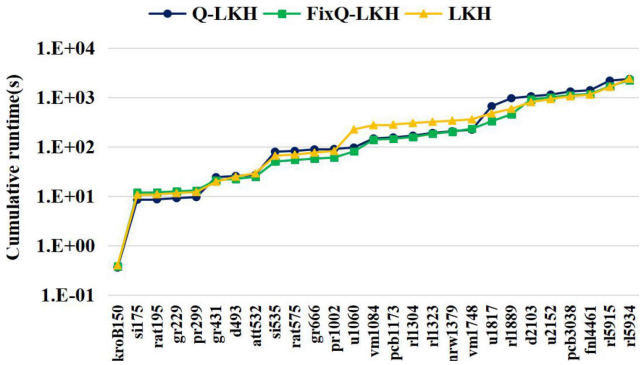
As shown in Figure 2(a), the solution quality of LKH is lower than that of FixQ-LKH, indicating that LKH can be improved by only replacing α-value with Q-value defined in Eq. 5 to select and sort candidate cities. The results of Q-LKH compared with FixQ-LKH in Figure 2(a) demonstrate the positive effects of reinforcement learning method (Q-learning) that learns to adjust Q-value during the iterations. In addition, as shown in Figure 2(b), there is almost no difference in efficiency among the three algorithms when solving the 27 *hard* instances.

Comparison on Reinforcement Strategies

Here we compare the effectiveness of the three reinforcement learning strategies (Q-learning, Sarsa and Monte Carlo) in reinforcing the LKH. The three variants of VSR-LKH reinforced only by Q-learning, Sarsa or Monte Carlo are called Q-LKH, SARSA-LKH and MC-LKH (M is fixed to be always 1, 2, or 3 in Algorithm 1 to obtain these three variants, respectively). Figure 3 shows their results coupled with the results of LKH, VSR-LKH and TD-LKH (a variant of VSR-LKH reinforced by Q-learning and Sarsa, M being 1 and 2 in turn in Algorithm 1) in solving the same set of 27

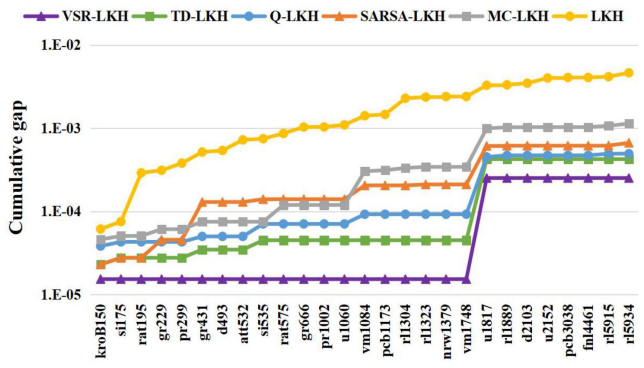


(a) Cumulative gap on the TSP instances

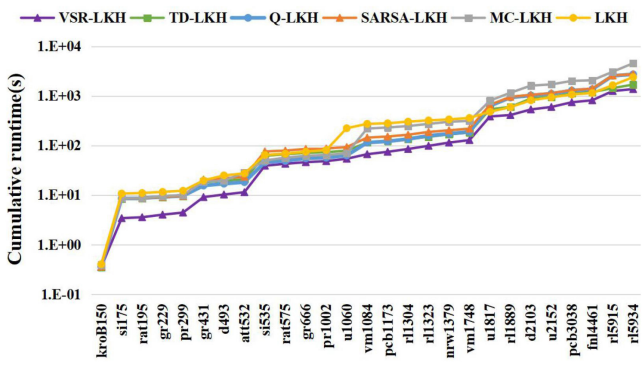


(b) Cumulative runtime on the TSP instances

Figure 2: Q-value evaluation on cumulative gap and runtime (instances are sorted in ascending order of problem size).



(a) Cumulative gap on the TSP instances



(b) Cumulative runtime on the TSP instances

Figure 3: Composite versus single reinforcement methods on cumulative gap and runtime.

hard instances in Figure 2. Each instance in Figure 3 is also calculated 10 times to get the average results.

As shown in Figure 3, incorporating any one of the three reinforcement learning methods (Q-learning, Sarsa or Monte Carlo) to the LKH can greatly improve the performance. And the three strategies are complementary in reinforcing LKH. For example, Q-LKH has better results than SARSA-LKH or MC-LKH in most instances in Figure 3(a), but in solving some instances like kroB150, si535 and rl5915, Q-LKH is not as good as SARSA-LKH.

Therefore, we proposed VSR-LKH to combine the advantages of the three reinforcement methods and leverage their complementarity. Since Q-learning has the best performance in reinforcing LKH, we order Q-learning first and then Sarsa. Monte Carlo is ordered in the last since it is not good at solving large scale TSP. The order of Q-learning and Sarsa in TD-LKH is the same as in VSR-LKH.

As shown in Figure 3, TD-LKH and VSR-LKH further improve the solution quality compared with the three reinforced LKHs with a single method, and the two composite reinforced LKHs are significantly better than the original LKH. In addition, though the results of MC-LKH in Figure 3(a) are the worst among single reinforcement algorithms when solving large scale TSP instances, the TD-LKH without Monte Carlo is not as efficient as VSR-LKH and is not as good as VSR-LKH in solving the instances such as gr431, si535 and u1817. Thus, adding the Monte Carlo method in VSR-LKH is effective and necessary.

Moreover, we randomly selected six instances (u574, u1060, u1817, fnl4461, rl5934 and rl11849) to compare the calculation results over the runtime of LKH and various variants of VSR-LKH. See detailed results in Appendix.

Final Comparison

We compare VSR-LKH with recent Deep Reinforcement Learning (DRL) algorithms for solving TSP, including S2V-DQN (Khalil et al. 2017) and Xing *et al.*'s method (2020). Table 1 compares VSR-LKH and the DRL algorithms in solving five *easy* instances with fewer than 100 cities that Xing *et al.* picked (Xing *et al.* only addressed instances with fewer than 100 cities). Column *Opt.* indicates the optimal solution of the corresponding instance. Each result of VSR-

NAME	Opt.	VSR-LKH	S2V-DQN	Xing *et al.*
eil51	426	**Opt.**	439	442
berlin52	7542	**Opt.**	**Opt.**	7598
st70	675	**Opt.**	696	695
eil76	538	**Opt.**	564	545
pr76	108159	**Opt.**	108446	108576

Table 1. Comparison of VSR-LKH and DRL algorithms on five *easy* instances (best results in bold).

NAME	Opt.	Method	Best	Average	Worst	Success	Time(s)	Trials
kroB150	26130	LKH	**Opt.**	26131.6	26132	2/10	0.40	128.4
		VSR-LKH	**Opt.**	26130.4	26132	8/10	0.35	61.5
d493	35002	LKH	**Opt.**	35002.8	35004	6/10	5.24	219.6
		VSR-LKH	**Opt.**	Opt.	Opt.	10/10	1.14	10.2
u1060	224094	LKH	**Opt.**	224107.5	224121	5/10	142.07	663.3
		VSR-LKH	**Opt.**	Opt.	Opt.	10/10	5.68	13.7
u1817	57201	LKH	**Opt.**	57251.1	57274	1/10	116.39	1817.0
		VSR-LKH	**Opt.**	57214.5	57254	7/10	256.79	766.9
rl1889	316536	LKH	316549	316549.8	316553	0/10	109.78	1889.0
		VSR-LKH	**Opt.**	Opt.	Opt.	10/10	26.30	91.9
d2103	80450	LKH	80454	80462.0	80473	0/10	216.48	2103.0
		VSR-LKH	**Opt.**	Opt.	Opt.	10/10	121.97	511.8
rl5915	565530	LKH	565544	565581.2	565593	0/10	494.81	5915.0
		VSR-LKH	**Opt.**	Opt.	Opt.	10/10	438.50	851.4
rl5934	556045	LKH	556136	556309.8	556547	0/10	753.22	5934.0
		VSR-LKH	**Opt.**	Opt.	Opt.	10/10	118.78	144.6
rl11849	923288	LKH	**Opt.**	923362.7	923532	2/10	3719.35	10933.4
		VSR-LKH	**Opt.**	Opt.	Opt.	10/10	1001.11	751.9
usa13509	19982859	LKH	**Opt.**	19983103.4	19983569	1/10	4963.52	13509.0
		VSR-LKH	**Opt.**	19982930.2	19983029	5/10	25147.63	11900.0

Table 2. Comparison of VSR-LKH and LKH on some *hard* instances, best results in bold (see full results in Appendix).

LKH is the average solution of 10 runs. Table 1 shows that VSR-LKH can always yield the optimal solution and is significantly better than the DRL algorithms.

Note that DRL algorithms are hard to scale to large scale problems. S2V-DQN provided results on 38 TSPLIB instances with the number of cities ranges from 51 to 318, but it didn't tackle larger instances due to the limitation of memory on a single graphics card. And the gap between the results of S2V-DQN and the optimal solution becomes larger when solving TSPs with more than 100 cities.

We then compare VSR-LKH and LKH on all the 111 TSPLIB instances. Table 2 shows detailed comparison on several typical *hard* instances with various scales (see full results on all *easy* and *hard* instances in Appendix). The two methods both run 10 times for each TSP instance in Table 2, and we compare the best solution, the average solution and the worst solution. Column *Success* indicates the number of times the algorithm obtains optimal solution, *Time* is the average calculation time of the algorithm and *Trials* is the average number of iterations of VSR-LKH and LKH.

As shown in Table 2, VSR-LKH outperforms LKH on every instance, especially on four *hard* instances, rl1889, d2103, rl5915 and rl5934, that LKH did not yield optimal solution. Note that VSR-LKH also achieves good results in solving large scale TSP instances with more than 10,000 cities (rl11849 and usa13509). Actually, VSR-LKH can yield optimal solution on almost all the 111 instances (a total of 107, except fl1577, d18512, pla33810 and pla85900) in 10 runs. And the average solution of VSR-LKH is also *Opt.* on most of the 111 instances (a total of 102, except kroB150, fl1577, u1817, usa13509, brd14051, d15112, d18512, pla33810, pla85900), indicating that VSR-LKH can always obtain optimal solution on these instances in 10 runs.

For the two super large instances, pla33810 and pla85900, we limit the maximum single runtime of LKH and VSR-LKH to 100,000 seconds due to the resource limit. And VSR-LKH can yield better results than LKH with the same runtime when solving these two instances (see Appendix).

In terms of efficiency, when solving the 111 TSPLIB instances, the average number of iterations of VSR-LKH is no more than that of LKH on 100 instances, especially for some *hard* instances (1060, rl1889, rl5934 and rl11849). Although VSR-LKH takes longer time than LKH in a single iteration because it needs to trade-off the exploration and exploitation in the reinforcement learning process, it can find the optimal solution with fewer *Trials* than LKH and terminate the iteration in advance. So the average runtime of VSR-LKH is less than that of LKH on most instances. In general, the proposed variable strategy reinforcement learning method greatly improves the performance of the LKH algorithm.

Conclusion

We combine reinforcement learning technique with typical heuristic search method, and propose a variable strategy reinforced method called VSR-LKH for the NP-hard traveling salesman problem (TSP). We define a Q-value as the metric for the selection and sorting of candidate cities, and change the method of traversing candidate set in selecting the edges to be added, and let the program learn to select appropriate edges to be added in the candidate set through reinforcement learning. VSR-LKH combines the advantages of three reinforcement methods, Q-learning, Sarsa and Monte Carlo, and further improves the flexibility and robustness of the proposed algorithm. Extensive experiments on public benchmarks show that VSR-LKH outperforms significantly the famous heuristic algorithm LKH.

VSR-LKH is essentially a reinforcement on the k-opt process of LKH. Thus, other algorithms based on k-opt could also be strengthened by our method. Furthermore, our work demonstrates the feasibility and privilege of incorporating reinforcement techniques with heuristics in solving classic combinatorial optimization problems. In future work, we plan to apply our approach in solving the constrained TSP and the vehicle routing problem.

Acknowledgments

This work is supported by National Natural Science Foundation (62076105).

References

Applegate, D. L.; Cook, W. J.; and Rohe, A. 2003. Chained Lin-Kernighan for Large Traveling Salesman Problems. *INFORMS Journal on Computing* 15(1): 82–92.

Barrett, T. D.; Clements, W. R.; Foerster, J. N.; and Lvovsky, A. 2020. Exploratory Combinatorial Optimization with Reinforcement Learning. In *AAAI*, 3243–3250.

Bello, I.; Pham, H.; Le, Q. V.; Norouzi, M.; and Bengio, S. 2017. Neural Combinatorial Optimization with Reinforcement Learning. In *ICLR*.

de O. da Costa, P. R.; Rhuggenaath, J.; Zhang, Y.; and Akcay, A. 2020. Learning 2-opt Heuristics for the Traveling Salesman Problem via Deep Reinforcement Learning. *CoRR* abs/2004.01608.

Fok, K.; Cheng, C.; Ganganath, N.; Iu, H. H.; and Tse, C. K. 2019. An ACO-Based Tool-Path Optimizer for 3-D Printing Applications. *IEEE Transactions on Industrial Informatics* 15(4): 2277–2287.

Gambardella, L. M.; and Dorigo, M. 1995. Ant-Q: A Reinforcement Learning Approach to the Traveling Salesman Problem. In *ICML*, 252–260.

Gao, W. 2020. New Ant Colony Optimization Algorithm for the Traveling Salesman Problem. *International Journal of Computational Intelligence Systems* 13(1): 44–55.

Held, M.; and Karp, R. M. 1970. The Traveling-Salesman Problem and Minimum Spanning Trees. *Operations Research* 18(6): 1138–1162.

Held, M.; and Karp, R. M. 1971. The traveling-salesman problem and minimum spanning trees: Part II. *Mathematical Programming* 1(1): 6–25.

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

Helsgaun, K. 2009. General *k*-opt submoves for the Lin-Kernighan TSP heuristic. *Mathematical Programming Computation* 1(2-3): 119–163.

Hougardy, S.; and Wilde, M. 2015. On the nearest neighbor rule for the metric traveling salesman problem. *Discrete Applied Mathematics* 195: 101–103.

Joshi, C. K.; Laurent, T.; and Bresson, X. 2019. An Efficient Graph Convolutional Network Technique for the Travelling Salesman Problem. *CoRR* abs/1906.01227.

Khalil, E. B.; Dai, H.; Zhang, Y.; Dilkina, B.; and Song, L. 2017. Learning Combinatorial Optimization Algorithms over Graphs. In *NIPS*, 6348–6358.

Lin, S. 1965. Computer solutions of the traveling salesman problem. *Bell Labs Technical Journal* 44(10): 2245–2269.

Lin, S.; and Kernighan, B. W. 1973. An Effective Heuristic Algorithm for the Traveling-Salesman Problem. *Operations Research* 21(2): 498–516.

Liu, F.; and Zeng, G. 2009. Study of genetic algorithm with reinforcement learning to solve the TSP. *Expert Systems with Applications* 36(3): 6995–7001.

Mladenovic, N.; and Hansen, P. 1997. Variable neighborhood search. *Computers & Operations Research* 24(11): 1097–1100.

Nagata, Y. 2006. Fast EAX Algorithm Considering Population Diversity for Traveling Salesman Problems. In *EvoCOP*, volume 3906, 171–182.

Nazari, M.; Oroojlooy, A.; Snyder, L. V.; and Takác, M. 2018. Reinforcement Learning for Solving the Vehicle Routing Problem. In *NeurIPS*, 9861–9871.

Pekny, J. F.; and Miller, D. L. 1990. An Exact Parallel Algorithm for the Resource Constrained Traveling Salesman Problem with Application to Scheduling with an Aggregate Deadline. In *ACM Conference on Computer Science*, 208–214.

Pesant, G.; Gendreau, M.; Potvin, J.; and Rousseau, J. 1998. An Exact Constraint Logic Programming Algorithm for the Traveling Salesman Problem with Time Windows. *Transportation Science* 32(1): 12–29.

Sanches, D. S.; Whitley, L. D.; and Tinós, R. 2017. Improving an exact solver for the traveling salesman problem using partition crossover. In *GECCO*, 337–344.

Shoma, M.; Daisuke, Y.; and Hiroyuki, E. 2018. Applying Deep Learning and Reinforcement Learning to Traveling Salesman Problem. In *ICCECE*.

Sun, R.; Tatsumi, S.; and Zhao, G. 2001. Multiagent reinforcement learning method with an improved ant colony system. In *SMC*, 1612–1617.

Sutton, R. S.; and Barto, A. G. 1998. *Reinforcement learning - an introduction*. Adaptive computation and machine learning. MIT Press.

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

Tinós, R.; Helsgaun, K.; and Whitley, L. D. 2018. Efficient Recombination in the Lin-Kernighan-Helsgaun Traveling Salesman Heuristic. In *PPSN XV*, volume 11101, 95–107.

Vinyals, O.; Fortunato, M.; and Jaitly, N. 2015. Pointer Networks. In *NIPS*, 2692–2700.

Wu, Y.; Song, W.; Cao, Z.; Zhang, J.; and Lim, A. 2019. Learning improvement heuristics for solving the travelling salesman problem. *arXiv preprint arXiv:1912.05784* .

Wunder, M.; Littman, M. L.; and Babes, M. 2010. Classes of Multiagent Q-learning Dynamics with epsilon-greedy Exploration. In *ICML*, 1167–1174.

Xing, Z.; Tu, S.; and Xu, L. 2020. Solve Traveling Salesman Problem by Monte Carlo Tree Search and Deep Neural Network. *CoRR* abs/2005.06879.