A Game-Theoretic Approach for Improving Generalization Ability of TSP Solvers

Anonymous authors
Paper under double-blind review

Abstract

In this paper, we shed new light on the study of how to improve the generalization ability of deep learning-based solvers for the Traveling Salesman Problem (TSP). We build a two-player zero-sum game between a trainable solver and a task generator, where the solver aims to solve instances provided by the generator, and the generator aims to generate increasingly difficult instances for the solver. Grounded in the *Policy Space Response Oracle* (PSRO) framework, our two-player framework allows us to obtain a behaviourally diverse population of powerful solvers over which we utilise a model mixing method to combine these solvers and achieve strong generalization ability on various tasks. Experimentally, we achieve the state-of-the-art results on a general TSP instance generation method over which the performance of other deep learning-based methods degenerates vastly. On realistic instances from TSPLib we approximately attain a **12%** improvement over the base model. Furthermore, we empirically illustrate as the solvers' performance improves, the obtained strategy's exploitability keeps decreasing showing gradual convergence to the Nash equilibrium.

1 Introduction

Deep learning-based methods for solving combinatorial optimization problems have attracted a great amount of attention recently due to its ability to capture the certain intrinsic structures by training over millions of instances (Dai et al., 2017). Because of the efficiency of the forward computation of neural networks, they are particularly useful in comparison to traditional methods when performing inference, especially on large-scale problems. As a consequence, it is promising to train offline deep learning-based solvers, and to implement them in an online scenario. There are two key factors in realising this:

1. The model should have excellent performance in known scenarios;
2. The model can handle various unknown cases, i.e. must have the ability to generalise.

Various methods have been proposed to achieve point 1 by training and testing solely on instances from the same distribution (Dai et al., 2017; Kool et al., 2018; Wu et al., 2021; Kool et al., 2021). For point two, researchers have also added the testing of their models on real-world instances (e.g.instances from TSPLIB (Reinelt, 1991)). However, the performance of most methods on these unseen situations are not competitive as that in their favorable cases. In addition, these TSPLIB instances are only an infinitesimal part during online usage, so the generalization ability has become the desiderata for deep learning-based solvers.

The generalization ability of a solver concerns its performance on various data distributions. In this work, we note that we can abstract the generalization problem into a two player game: player 1 aims to train the solver to perform well on the distribution chosen by player 2, and player 2 aims to generate distributions where player 1 performs poorly. It is obvious that player 2 has an essentially infinitely-sized policy set as there are an enormous amount of candidate distributions. In this view, previous works solely focus on the uniform distribution which is only one of the many policies available in the policy set of player 2, helping to explain poor generalisation ability. It's intuitive that better generalization ability can be obtained by exploiting the policy set of player 2, and PSRO (Lanctot et al., 2017) provides a methodology to achieve this goal.

In this paper, we aim to improve the generalization ability of deep learning-based solvers from a game theoretic perspective. Specifically, we build a two player zero-sum game for a set of solvers and data generators. The purpose of the solvers is to perform well on the given data distributions, and the objective of the data generators is to explore the distributions where the solvers perform unsatisfactorily. Under the framework of PSRO (Lanctot et al., 2017), we can generate a population of data distribution where the deep learning-based solvers fail and train the solver to overcome its own weaknesses by iteratively obtaining oracles for both players. In this way, the deep learning-based solvers can automatically find the hard-to-solve distributions and train itself on these distributions in the following PSRO iterations. Additionally, we analyse the game-theoretic properties of the meta-game payoff tables generated by the population of solvers and data generators, which provides us a higher level view on the generalization ability of our framework. We focus on the TSP, a typical combinatorial problem which is widely used in the real-world, to demonstrate the ability of our method. For our population of solvers we select a powerful deep learning-based solver (Wu et al., 2021) as our base solver due to its excellent performance on these problems. For the data generators, we propose an attacked model: we add *learnable* perturbations on top of uniformly generated data to obtain a distribution where the current solver performs poorly, aiming to find weaknesses with the solver[1]. Otherwise, under proper settings, any combinatorial optimization problems, any existing deep learning-based solvers and any data generating methods can fit into this general framework.

Overall, the contributions of our work are as follows:

- We are the first to study the generalization ability of the given deep learning-based solvers from the game perspective, and propose a general framework to obtain powerful combined solvers which can incorporate with minimal changes any deep learning-based solvers;
- For TSP, we propose an 'attacking' method by adding *learnable* perturbations on uniform data to exploit the distributions where the solvers do not perform well, which will give us a intuitive view on the weakness of given solvers;
- We introduce a mixing-model by combining a population of solvers so that we can make full use of the obtained solvers. Under a fair comparison metric, our mixing-model attains the state-of-the-art results ob both generated data and real world data.
- We study the exploitability in the game between the solver and data generator which gives substantial insights about the game. The results show that the improvement of the population of solvers is consistent with a decrease in exploitability, which suggests that the obtained strategies are gradually approaching the Nash Equilibrium.

2 Related Work

Deep Learning for Combinatorial Optimization. Pointer Networks (Vinyals et al., 2015) were the first attempt to solve combinatorial optimization problems through deep learning. They solve the TSP by training a pair of RNN-based encoders and decoders to output a permutation over its inputs. Then (Bello et al., 2016) used a similar model but trained the model with reinforcement learning instead of the Supervised learning used in Pointer Networks, which only needs the tour length of the given instance as the reward signal. Based on Transformers (Vaswani et al., 2017), Attention Model (Kool et al., 2018) uses the attention mechanism, which is similar to Graph Attention Networks (Veličković et al., 2017), to encode the node representation and decode the strategy for solving various problems. (Lu et al., 2019) propose the "Learn to Improve"(L2I) model which iteratively refines the solution with improvement operators and jumps out of local optima with perturbation operators. Similarly, (Wu et al., 2021) propose a reinforcement learning framework to learn the improvement heuristics for the TSP and Capacitated Vehicle Routing Problem (CVRP), and achieves excellent results both on random generated instances and some real world instances. The latest research (Kool et al., 2021; Fu et al., 2020) focuses on pre-training models and constructing heatmaps to guide the search method. Although deep learning or reinforcement learning based methods have achieved notable progress in various combinatorial optimization problems, some essential issues have emerged, such as: unsatisfactory results on large-scale problems, computational effort during training or generalization ability. (Bengio et al., 2020) provides a comprehensive view on various problems in this area.

Meta-Game. In meta-game analysis(Wellman, 2006; Tuyls et al., 2018), traditional solution concepts (e.g., NE or α-Rank (Omidshafiei et al., 2019)) can be computed in a more scalable manner as the number of 'higher-level' strategies in the meta-game is usually far smaller than the number of atomic actions of the underlying game. Various methods are proposed to solve meta-games, with the original

[1]We show these results in the Appendix A.5.

work being Double Oracle (McMahan et al., 2003) which expands player's policy sets by adding corresponding best responses iteratively. PSRO (Lanctot et al., 2017) generalises Double Oracle by introducing RL to obtain an approximate best response. PSRO$_{rN}$ (Balduzzi et al., 2019) and (Nieves et al., 2021) incorporate diversity seeking into PSRO and Pipeline-PSRO (McAleer et al., 2020) aims to improve training efficiency by training multiple best responses in parallel. Due to the PPAD-Hard (Daskalakis et al., 2009) complexity of computing Nash Equilibrium, α-PSRO (Muller et al., 2019) and α^α-Rank (Yang et al., 2019) replace NE with α-Rank (Omidshafiei et al., 2019) which can solve general-sum games in polynomial-time, and (Feng et al., 2021) use neural networks to compute the meta-distribution.

3 Preliminary

Normal Form Game (NFG) can be described as a tuple (Π, U^Π, n) where n is the number of players, $\Pi = (\Pi_1, \Pi_2, ..., \Pi_n)$ is the joint policy set and $U^\Pi = (U_1^\Pi, U_2^\Pi, ...U_n^\Pi) : \Pi \to \mathcal{R}^n$ is the utility table for each joint policy. We call a game symmetric if all the players have the same policy set ($\Pi_i = \Pi_j, i \neq j$) and payoff structures such that players are interchangeable.

Best Response is the strategy which attains the best excepted performance against a fixed opponent strategy. $\sigma_i^* = \text{br}(\Pi_{-i}, \sigma_{-i})$ is the best response to σ_{-i} if:

$$U_i^\Pi(\sigma_i^*, \sigma_{-i}) \geq U_i^\Pi(\sigma_i, \sigma_{-i}), \forall i, \sigma_i \neq \sigma_i^*$$

Nash Equilibrium for an NFG is a strategy profile $\sigma^* = (\sigma_1^*, \sigma_2^*, ..., \sigma_n^*)$ such that:

$$U_i^\Pi(\sigma_i^*, \sigma_{-i}^*) \geq U_i^\Pi(\sigma_i, \sigma_{-i}^*), \forall i, \sigma_i \neq \sigma_i^*$$

Intuitively, no player has an incentive to deviate from their current strategy if all players are playing their respective Nash equilibrium strategy.

Exploitability measures how much expected utility a best-response opponent can achieve above the game value (Davis et al., 2014). In a two-player zero-sum game, the average exploitability of a strategy profile $\sigma = (\sigma_1, \sigma_2)$ is defined as follows:

$$exploit(\sigma) = \frac{1}{2}(U_1^\Pi(\text{br}(\sigma_2), \sigma_2) + U_2^\Pi(\sigma_1, \text{br}(\sigma_1))) \tag{1}$$

Instance represents an individual sample of a given type of combinatorial optimization problem. For example, given the two-dimensional coordinates of n points, finding the shortest tour that traverses all points can be seen as an instance of a TSP. In the following, we denote an instance by $\mathcal{I}$. Generally, we can consider the instances to be solved as coming from a certain distribution $\mathbf{P}_\mathcal{I}$.

Optimal gap measures the quality of a given solver compared with an Oracle solver. Given an instance $\mathcal{I}$ and a solver $S : \{\mathcal{I}\} \to \mathbf{R}$, the optimal gap is defined as:

$$g(S, \mathcal{I}, \text{Oracle}) = \frac{S(\mathcal{I}) - \text{Oracle}(\mathcal{I})}{\text{Oracle}(\mathcal{I})} \tag{2}$$

where Oracle$(\mathcal{I})$ gives the (possible) true optimal value of the instance. Furthermore, the expected optimal gap of the given instance distribution $\mathbf{P}_\mathcal{I}$ and an Oracle is defined as:

$$G(S, \mathbf{P}_\mathcal{I}, \text{Oracle}) = \mathbf{E}_{\mathcal{I} \sim \mathbf{P}_\mathcal{I}} g(S, \mathcal{I}, \text{Oracle}). \tag{3}$$

4 Method

In this section, we will give our formulation for solving the Traveling Salesman Problem (TSP) at the meta-level. Assume there are two players in the meta-game, one is the solver selector (SS) and the other is the data generator (DG). $\Pi_{\text{SS}} = \{S_i | i = 1, 2, ...\}$ is the policy set for solver selector which contains a set of candidate solvers for the given TSP, and $\Pi_{\text{DG}} = \Pi_N \times \Pi_C = \{\mathbf{P}_{\mathcal{I},i} = (\mathbf{P}_{N,i}, \mathbf{P}_{C,i}) | i = 1, 2, ...\}$ is the policy set for the data generator, where Π_N is the policy set of the problem scale (i.e. the number of nodes that need to be generated) and Π_C denotes the policy set used to generate two-dimension coordinate points. Therefore, an instance distribution

Algorithm 1 PSRO for Combinatorial Optimization

```
Input: Initial joint policy sets for solver selector and data generator as $\Pi$. Compute utilities $U^{\Pi}$
for joint $\pi \in \Pi$. Initialize meta-strategies $\sigma_i$ = UNIFORM($\Pi_i$)
while epoch e in $\{1, 2, ...\}$ do
    Construct mixing distribution $\pi_{\text{mix}} = \sum_i \sigma^i_{\text{DG}} \pi^i_{\text{DG}}$ and train the oracle for solver selector $S^{'}$
    with gradient in Eq. 6
    for many episodes do
        Sample $S \sim \Pi_{\text{SS}}$ according to $\sigma_{\text{SS}}$
        Train Oracle $\mathbf{P}^{'}_{\mathcal{I}} = \text{br}(S)$ with gradient in Eq. 11
    end for
    Update policy set:$\Pi \leftarrow \Pi \cup \{(S^{'}, \mathbf{P}^{'}_{\mathcal{I}})\}$
    Compute missing entries in $U^{\Pi}$ from $\Pi$ and the meta-strategy $\sigma$ from $U^{\Pi}$
end while
Output meta-strategy $\sigma_{\text{SS}}$ and policy set $\Pi_{\text{SS}}$ to obtain mixing model by Eq. 12 or Eq. 13.
```

$\mathbf{P}_{\mathcal{I}} \in \Pi_{\text{DG}}$ comprises two parts: $\mathbf{P}_{\mathcal{I}} = (\mathbf{P}_N, \mathbf{P}_C)$ which is the distribution for the number of nodes N contained in each instance, and each coordinate is sampled from a two-dimensional distribution $\mathbf{P}_C$ independently.

Formally, we can formulate a two player zero-sum asymmetric NFG $(\Pi, \mathbf{U}^{\Pi}, 2)$ where $\Pi = (\Pi_{\text{SS}}, \Pi_{\text{DG}})$, $\mathbf{U}^{\Pi} : \Pi \rightarrow \mathbf{R}^{|\Pi_{\text{SS}}| \times |\Pi_{\text{DG}}|}$, $\mathbf{U}^{\Pi_{\text{SS}}}(\pi) = G(\pi, \text{Oracle})$ is the expected optimal gap under the joint policy $\pi = (S, \mathbf{P}_{\mathcal{I}}) \in \Pi$ as defined in Eq. 3 and $\mathbf{U}^{\Pi_{\text{DG}}}(\pi) = -\mathbf{U}^{\Pi_{\text{SS}}}(\pi) = G(\pi, \text{Oracle})$. Given $\mathbf{U}^{\Pi}$, we can determine a Nash Equilibrium $\sigma^* = (\sigma^*_{\text{SS}}, \sigma^*_{\text{DG}})$ as the meta-strategy which satisfies:

$$\min_{\sigma_{\text{SS}} \in \Delta(\Pi_{\text{SS}})} \max_{\sigma_{\text{DG}} \in \Delta(\Pi_{\text{DG}})} \mathrm{E}_{\pi \sim (\sigma_{\text{SS}}, \sigma_{\text{DG}})} G(\pi, \text{Oracle}). \tag{4}$$

Following the PSRO framework, given the meta-strategy $\sigma = (\sigma_{\text{SS}}, \sigma_{\text{DG}})$ we train an Oracle $S^{'}$ for the Solver to give the best response to the data generator's meta strategy σ_{DG}, and an Oracle $\mathbf{P}^{'}_{\mathcal{I}}$ for the data generator which represents a data distribution where σ_{SS} performs poorly. Given these oracles, we update the joint policy set $\Pi^{'} = \Pi \cup (S^{'}, \mathbf{P}^{'}_{\mathcal{I}})$ and the meta-game $\mathbf{U}^{\Pi^{'}}$ according to the new joint policy set $\Pi^{'}$. As a result, we enhance the abilities of both populations by adding a better Solver, and a more difficult-to-solve instance distribution. This expansion of the policy set allows for not only an improvement in the Solver's ability to handle instances from different distributions, but also explores the instance distributions that are difficult to solve. In line with our objective, this process leads to a population of powerful solvers which have diverse abilities on various distributions, and we can gather them together to improve performance through a combined Solver model. The general algorithm framework can be seen in Alg. 1.

To this end, we must address four issues:

- How to obtain the meta-strategy σ;
- How to train Oracles for both players;
- How to evaluate the utilities U^{Π};
- How to combine the ability of the obtained solvers.

In the following parts, we will elaborate the details about these issues and the pipeline is shown in Fig. 1

4.1 Meta-Strategy Solvers

A variety of meta-strategy solvers have been proposed for different game scenarios: for two-player (population) zero-sum games, solving Nash Equilibrium is a desirable and the simplest way to get meta-strategy; replicator-dynamics (Taylor & Jonker, 1978) can reveal the process and potential of how a strategy profile evolves in the midst of others. Both NE and replicator-dynamics are suitable for two-player games. And for the game which contains more than two players, α-Rank (Omidshafiei et al., 2019) is a powerful evaluation method which can scale tractably in the number of players, in the type of interactions and the type of empirical games. In this paper, we use the NE of the meta

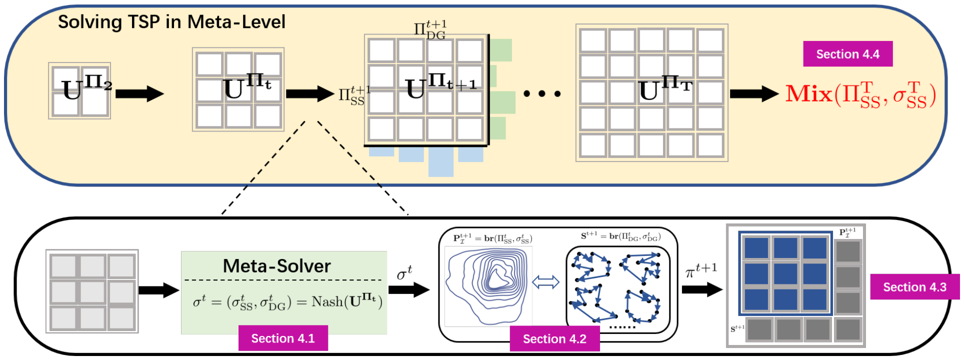

Solving TSP in Meta-Level
$\mathbf{U}^{\Pi_2}$
$\mathbf{U}^{\Pi_t}$
Π^{t+1}_{SS}
Π^{t+1}_{DG}
$\mathbf{U}^{\Pi_{t+1}}$
$\mathbf{U}^{\Pi_T}$
Section 4.4
$\mathbf{Mix}(\Pi^T_{SS}, \sigma^T_{SS})$
Meta-Solver
$\sigma^t = (\sigma^t_{SS}, \sigma^t_{DG}) = \text{Nash}(\mathbf{U}^{\Pi_t})$
Section 4.1
σ^t
$\mathbf{P}^{t+1}_{\mathcal{I}} = \mathbf{br}(\Pi^t_{SS}, \sigma^t_{SS})$
$\mathbf{S}^{t+1} = \mathbf{br}(\Pi^t_{DG}, \sigma^t_{DG})$
Section 4.2
π^{t+1}
$\mathbf{P}^{t+1}_{\mathcal{I}}$
$\mathbf{S}^{t+1}$
Section 4.3

Figure 1: Pipeline of solving combinatorial optimization problems in meta-level: At PSRO loop t, we first use meta-solver to compute the meta-strategy σ_t given the meta-table $\mathbf{U}^{\Pi_t}$ and then training best response $(\mathbf{S}^{t+1}, \mathbf{P}^{t+1}_{\mathcal{I}})$ based on current policy set and meta-strategy (Π^t, σ^t). Finally we get a new meta-table $\mathbf{U}^{\Pi_{t+1}}$ according to the new obtained policy and algorithm process transfers to the next loop.

game as the meta-strategy for simplicity, but various meta-solvers can be used w.r.t the corresponding meta-game constructed by the specific combinatorial optimization problem.

4.2 Oracle Training

We now provide details on training an oracle in the TSP setting, with more detailed derivations in Appendix A.1. Here we represent the Solvers in Π_{SS} as S_θ and the instance distributions in Π_{DG} as $\mathbf{P}_{\mathcal{I},\gamma} = (\mathbf{P}_{N,\gamma_N}, \mathbf{P}_{C,\gamma_C})$ where θ and γ are the trainable parameters.

Training oracle for the Solver selector. Given σ_{DG}, the oracle training objective is:

$$\min_\theta L_{\text{SS}}(\theta) = \mathbf{E}_{\mathbf{P}_\mathcal{I}\sim\sigma_{\text{DG}}} G(S_\theta, \mathbf{P}_\mathcal{I}, \text{Oracle}). \tag{5}$$

The gradient of this objective is:

$$\nabla_\theta L_{\text{SS}}(\theta) \quad = \mathbf{E}_{\mathbf{P}_\mathcal{I}\sim\sigma_{\text{DG}}} \mathbf{E}_{N\sim\mathbf{P}_N} \mathbf{E}_{x_1,\ldots,x_N\sim\prod_{i=1}^N \mathbf{P}_C} \frac{\nabla_\theta S_\theta(x_1,\ldots,x_N)}{\text{Oracle}(x_1,\ldots,x_N)}. \tag{6}$$

Training oracle for the data generator. Given σ_{SS}, the oracle training objective is:

$$\max_\gamma L_{\text{DG}}(\gamma) = \mathbf{E}_{S\sim\sigma_{\text{SS}}} G(S, \mathbf{P}_{\mathcal{I},\gamma}, \text{Oracle}). \tag{7}$$

For the gradient of the data generator oracle, we require a better understanding of the parameterization. For Π_N, we fix the optional problem scales $\mathcal{N} = \{N_1, N_2, \ldots\}$, and let $\mathbf{P}_{N,\gamma_N} \in \Pi_N$ be a parameterized discrete distribution over $\mathcal{N}$, that is, $\gamma_{N,i} = \mathbf{P}_{N,\gamma_N}(N_i)$.

We note that it is easy to attack the solver by adding noise to the given instances[2], so we take potential attacks into consideration to improve the robustness of the solver. We achieve this by utilising an *attacked* distribution where instances sampled from a uniform distribution are perturbed by Gaussian noise. Formally, we first sample $\mathcal{I} \sim \text{U}(0,1)$, then we use an attack generator f_{γ_C} parameterized by neural networks to generate the variation of Gaussian distributions:

$$\Sigma = f_{\gamma_C}(\mathcal{I}) \tag{8}$$

where the shape of Σ is the same as $\mathcal{I}$, that is, if $\mathcal{I}$ contains N two dimension coordinates, $\Sigma \in \text{R}^{N\times 2}$ and we attack the instance $\mathcal{I}$ by $\tilde{\mathcal{I}}_{i,j} = \mathcal{I}_{i,j} + \epsilon$ where $\epsilon \sim \text{N}(0, \Sigma_{i,j})$, we denote the final attacked distribution by $\mathbf{P}_{C,\gamma_C}$. Our objective is therefore to find the optimal parameter $\gamma^* = (\gamma^*_C, \gamma^*_N)$.

The gradient w.r.t. γ_C is:

$$\begin{aligned}\nabla_{\gamma_C} L_{\text{DG}}(\gamma) = \mathbf{E}_{S\sim\sigma_{\text{SS}}} \mathbf{E}_{N\sim\mathbf{P}_{N,\gamma_N}} \mathbf{E}_{x_1,..,x_N\sim\prod_{i=1}^N \mathbf{P}_{C,\gamma_C}} \cdot \\ \nabla_{\gamma_C}\Big(\sum_{i=1}^N \log \mathbf{P}_{C,\gamma_C}(x_i)\Big) g(S, (x_1, \ldots x_N), \text{Oracle}).\end{aligned} \tag{9}$$

[2]Demonstrations are shown in Appendix A.5

An extra computation is required for the log-probability in Eq. 9 which we leave to Appendix A.2. The gradient w.r.t. γ_N is:

$$\nabla_{\gamma_N} L_{\text{DG}}(\gamma) = \mathbf{E}_{S\sim\sigma_{\text{SS}}} \mathbf{E}_{N\sim\mathbf{P}_{N,\gamma_N}} \nabla_{\gamma_N} (\log \mathbf{P}_{N,\gamma_N}(N)) \cdot \\ \mathbf{E}_{x_1,..,x_N \sim \prod_{i=1}^{N} \mathbf{P}_{C,\gamma_C}} g(S, (x_1, ...x_N), \text{Oracle}). \tag{10}$$

Overall, we get the gradient of the training oracle for the data generator:

$$\nabla_\gamma L_{\text{DG}}(\gamma) = \begin{pmatrix} \nabla_{\gamma_C} L_{\text{DG}}(\gamma) \\ \nabla_{\gamma_N} L_{\text{DG}}(\gamma) \end{pmatrix} \tag{11}$$

4.3 Evaluation

Given the joint policy set Π, we can compute the elements in matrix U^Π by approximating Eq. 3:

$$u_{S,\mathbf{P}_\mathcal{I}} = G(S, \mathbf{P}_\mathcal{I}, \text{Oracle}) \approx \frac{1}{M} \sum_{i=1}^{M} g(S, \mathcal{I}_i, \text{Oracle}).$$

where $S \in \Pi_{SS}, \mathbf{P}_\mathcal{I} \in \Pi_{DG}$, which means the expected optimal gap of solver S given instance distribution $\mathbf{P}_\mathcal{I}$.

4.4 Combining the Solver population

After the training of PSRO, we obtain a population of solvers which have diverse abilities for different distributions. It's promising to combine these solvers to obtain a best solver. There are several works discussing how to mix policies (Smith et al., 2020; 2021). Different from Q-Mxing (Smith et al., 2020), we choose to combine the solvers with solver selector's meta-strategy in this work. Due to the use of solver selector's meta-strategy (nash strategy) to combine solvers, we can guarantee a conservative choice to deal with any instances under the assumption that these instances' distribution can be generated by data generator's policy set. In the view of game theory, the nash strategy is always a not too bad choice whatever the opponent's strategy is and even though the opponent is not rational. To some extent, the conservativeness of nash weights has the accordance to the meaning of generalization ability. What's more, different from Q-mixing (Smith et al., 2020) which supports value-based methods only, our mixing-method has no prior assumptions on certain RL algorithm so it can be used for both value-based and policy-based solvers.

Formally, for the value-based RL solvers, we can weight the corresponding Q values to get the mixing Q value as the combined model:

$$Q_{\text{mix}}(s, a) = \sum_{\pi\in\Pi_{\text{SS}}} \sigma^*_{\text{SS}}(\pi) Q_\pi(s, a) \tag{12}$$

and then using $Q_{\text{mix}}(s, a)$ to make decisions. For the policy-based RL solvers, we directly obtain the mix policy probability by:

$$\pi_{\text{mix}}(a|s) = \sum_{\pi\in\Pi_{\text{SS}}} \sigma^*_{\text{SS}}(\pi)\pi(a|s) \tag{13}$$

5 Experiments

In this section, we show results on As the general setting used in previous works, we investigate the performance on problem instances of $n = 20, 50, 100$. Specific training settings are listed in Appendix A.3.

5.1 Settings

Data normalization. We only consider the TSP instances within $[0, 1] \times [0, 1]$ and we normalize the given instance by min-max normalization:

$$\mathcal{I}_{norm} = \text{Norm}(\mathcal{I}) = \frac{\mathcal{I} - \min\{\mathcal{I}\}}{\max\{\mathcal{I}\} - \min\{\mathcal{I}\}} \tag{14}$$

Table 1: Our model vs baselines. The gap % is w.r.t. the best value across all methods.

		$n = 20$			$n = 50$			$n = 100$		
	Method	Obj.	Gap	Time	Obj.	Gap	Time	Obj.	Gap	Time
	Concorde	3.43	0.00%	(6s)	4.99	0.00%	(1m)	6.20	0.00%	(3m)
	LKH3	3.43	0.00%	(2s)	4.99	0.00%	(27s)	6.20	0.00%	(3m)
	Gurobi	3.43	0.00%	(1s)	4.99	0.00%	(19s)	6.20	0.00%	(4m)
	Farthest Insertion	3.52	2.62%	(1s)	5.26	5.41%	(1s)	6.66	7.41%	(2s)
	Random Insertion	3.59	4.66%	(0s)	5.38	7.82%	(1s)	6.81	9.84%	(1s)
	Nearest Insertion	3.82	11.37%	(1s)	5.87	17.64%	(2s)	7.45	20.16%	(2s)
	AM(gr.)	3.45	0.58%	(2s)	5.12	2.61%	(4s)	6.71	8.23%	(8s)
TSP	LIH(T=1000)	3.69	7.72%	(16s)	5.07	1.72%	(33s)	6.72	8.48%	(63s)
	LIH(FS)(T=1000)	**3.43**	**0.12%**	(18s)	**5.06**	**1.68%**	(34s)	**6.47**	**4.43%**	(67s)
	LIH(FT)(T=1000)	**3.43**	**0.04%**	(50s)	**5.07**	**1.70%**	(64s)	**6.45**	**4.00%**	(2m)
	AM(sampling)	3.43	0.11%	(6s)	5.03	0.95%	(29s)	7.22	16.45%	(2m)
	LIH(T=3000)	3.62	5.54%	(46s)	5.03	0.92%	(95s)	6.58	6.24%	(3m)
	DPDP(bs=10K)	3.43	**0.00%**	(5s)	5.03	1.00%	(3m)	6.86	10.66%	(12m)
	LIH(FS)(T=3000)	**3.43**	0.04%	(54s)	**5.03**	**0.88%**	(100s)	**6.37**	**2.77%**	(3m)
	LIH(FT)(T=3000)	**3.43**	**0.00%**	(2m)	**5.03**	**0.89%**	(3m)	**6.36**	**2.50%**	(6m)

where $\min\{\mathcal{I}\}$ means the minimum scalar value in the TSP instance coordinates and $\max\{\mathcal{I}\}$ is similar. For TSP, it's easy to verify that the instance after normalization has the same optimal solution (not optimal value) as the un-normalized one.

Data generation. Unlike previous works which train and test models in the same distribution (usually uniform), we make validations on the distribution where the model never met during training. We generate data by randomly sampling $x \in \mathrm{R}^2$ from the unit square and sampling $y \in \mathrm{R}^2$ from $\mathrm{N}(\mathbf{0}, \Sigma)$ where $\Sigma \in \mathrm{R}^{2\times 2}$ is a diagonal matrix whose elements are sampled from $[0, \lambda]$ and $\lambda \sim \mathrm{U}(0, 1)$. Then a two-dimension coordinate is generated by $z = x + y$ and we can get any scale of TSP by performing these steps repeatedly. We then get the generated instance followed by the normalization step in Eq. 14. We sample 1000 instances which comprises 10 groups data generated by different λ and then report the relevant results on the generated dataset.

Baselines. In this paper, we use a typical RL solvers: (Wu et al., 2021) which we call **LIH** as our base model and we use two kinds of training paradigm: training from scratch **LIH(FS)** and fine-tune **LIH(FT)**, specific settings can be seen in Appendix A.3. We compare our model with Concorde, LKH3 (Helsgaun, 2017) and some heuristic methods, as well as typical deep learning-based methods: AM (Kool et al., 2018), (Wu et al., 2021) and the new state-of-the-art method DPDP (Kool et al., 2021). All experiments are trained and executed with one single GPU (RTX3090) and CPU (i9-10900KF).

5.2 RESULTS

Results on generated data. We first compare **LIH(FS)** on the generated data which never met during training. In Table 1, we can see that the performance of deep learning based methods trained on uniform distribution are all degraded a lot when dealing with instances from unseen distribution. However, the model obtained from the PSRO framework performs well and get the state-of-the-art results among the deep learning methods and classic heuristic methods. Notice that we do not tune any hyperparameters in the original solver and all these improvements are just based on the changing of training paradigm under PSRO. However, due to the use of mixing-strategy, the time consuming grows linearly compared with the base solver because of the extra feed-forward computation, which can be seen as a trade off between solution quality and running time.

Results on real world problems. We also test **LIH(FS)** and **LIH(FT)** on the real TSP problems from TSPLib (Reinelt, 1991) in Table 2. We keep the same settings as (Wu et al., 2021) with T= 3000. And the testing results about OR-Tools, LIH are directly taken from the Table 5 in (Wu et al., 2021).

From the results in Table 2, **LIH(FS)** and **LIH(FT)** has a prominent performance compared with the base model LIH by a 12% improvement in average. What's more, our model has the best performance among deep learning based methods in the most cases and for a part of instances: eil51, pr124, rd100, pr76, kroB150, u159, eil101, kroC100, eil76, kroB100, kroE100, bier127, our method achieves better performances than OR-Tools.

Table 2: Results on TSPlib Instances. The underlined and bold figures mean achieving the best results among all methods (including OR-Tools) and all deep learning-based methods respectively.

Instance	Opt.	OR-Tools	AM (N=1,280)	AM (N=5,000)	LIH (T=3,000)	**LIH(FS)** (T=3,000)	**LIH(FT)** (T=3,000)
pr226	80,369	82,968	91,765	89,895	97,348	**84,088**	**84,088**
ts225	126,643	128,564	139,180	139,871	158,748	**136,699**	**136,699**
kroD100	21,294	21,636	23,582	23,336	24,771	**21,828**	22,300
eil51	426	436	435	434	438	**429**	**429**
kroA100	21,282	21,448	25,163	24,450	25,196	**21,703**	22,289
pr264	49,135	51,954	66,222	66,213	65,946	**55,312**	**55,312**
pr152	73,682	75,834	82,186	84,104	85,616	**76,389**	**76,389**
gil262	2,378	2,519	2,708	2,679	2,963	**2,615**	**2,615**
rat99	1,211	1,232	1,459	1,345	1,419	**1,239**	1,248
kroA150	26,524	27,592	29,990	29,826	31,244	**28,628**	**28,628**
lin105	14,379	14,824	24,239	22,683	18,194	**15,372**	**15,372**
pr124	59,030	62,519	62,750	61,996	66,010	**61,645**	**61,645**
st70	675	683	691	**690**	706	696	693
a280	2,579	2,713	3,247	3,236	2,989	**2,819**	**2,819**
rd100	7,910	8,189	8,180	8,048	7,915	**8,036**	8,160
pr136	96,772	102,213	103,035	**102,496**	105,618	104,429	104,429
pr76	108,159	111,104	111,598	111,924	109,668	109,418	**108,495**
kroA200	29,368	29,714	34,866	34,556	35,958	**31,450**	**31,450**
kroB200	29,437	30,516	35,003	35,387	36,412	**31,656**	**31,656**
pr107	44,303	45,072	83,926	62,392	53,056	**45,288**	**45,288**
kroB150	26,130	27,572	28,894	28,864	31,407	**27,418**	**27,418**
u159	42,080	45,778	45,394	44,581	51,327	**43,376**	**43,376**
berlin52	7,542	7,945	9,759	9,831	8,020	7,653	**7,544**
rat195	2,323	2,389	2,783	2,697	2,913	**2,600**	**2,600**
d198	15,780	15,963	77,722	70,692	17,962	**16,501**	**16,501**
eil101	629	664	656	656	658	**642**	656
pr144	58,537	59,286	65,493	66,338	71,006	**62,522**	**62,522**
pr299	48,191	48,447	340,135	299,597	59,786	**51,726**	**51,726**
kroC100	20,749	21,583	22,586	22,896	25,343	**21,079**	21,255
tsp225	3,916	4,046	5,004	4,790	4,701	**4,262**	**4,262**
eil76	538	561	558	557	575	**548**	**548**
kroB100	22,141	23,006	24,340	23,987	26,563	**22,855**	23,677
kroE100	22,068	22,598	22,895	22,716	26,903	**22,532**	22,898
ch150	6,528	6,729	6,827	**6,787**	7,916	6,866	6,866
bier127	118,282	122,733	130,513	128,150	142,707	**127,520**	**127,520**
ch130	6,110	6,284	6,311	6,302	7,120	**6,495**	**6,495**
Avg. Gap (%)	0	3.46	42.96	36.86	17.12	**5.13**	5.49

6 DISCUSSION

In this section, we will provide analysis about property of meta-game in TSP, usage of population solvers and effects of different initial solutions provided for the model.

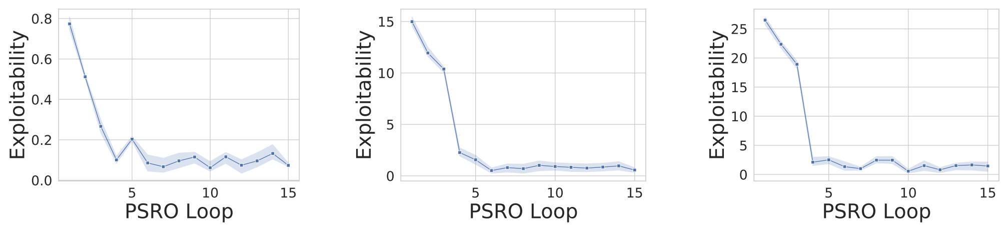
6.1 META GAME ANALYSIS

To prove the rationality of our method in the aspect of game theory, we provide the meta analysis about the meta-payoff according to the exploitability in Eq. 1. Specifically, we calculate the the exploitability of the meta-game obtained from the training under the PSRO framework. For demonstration, we train 15 PSRO loops for TSP20, TSP50, TSP100 respectively in total and compute the exploitability of the new obtained strategy at each PSRO loop. Results can be seen in Fig. 2. Owing to the use of Nash Equilibrium as meta-solver, these results demonstrate the validity of our method. Specifically, as the training goes on, we can get a strategy which gets monotonically close to the Nash Equilibrium as shown in Fig. 2(a) to 2(c). What's more, seeing the pair of Fig. 2(a) and Fig. 2(d), Fig. 2(b) and Fig. 2(e), Fig. 2(c) and Fig. 2(f), we can see there exixts the accordance between the decrease of the exploitability and the improvement of mixing-solver's performance.

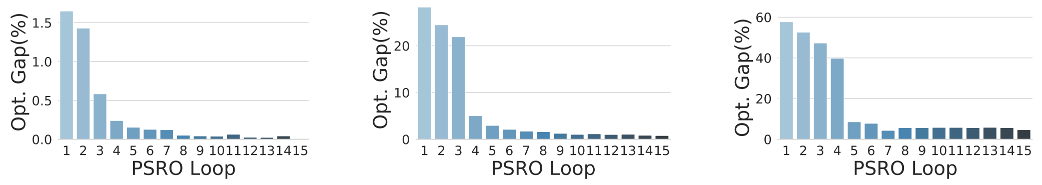
6.2 USAGE OF A POPULATION OF SOLVERS

In this part, we discuss the effects of different amounts of solvers used and different mixing weights for combining solvers. And all demonstrated results are obtained by **LIH(FS)** (T=1000). We use the instances generated by the method in Section 5 to make sure the generated data are never experienced during training periods. Here we consider three scenarios:

- Original: the whole population of solvers combined with solver selector's meta-strategy
- Uniform: the whole population of solvers combined uniformly
- Original-Partial: the two most powerful solvers judged by the solver selector's meta strategy and combined with the normalized probability in the meta-strategy

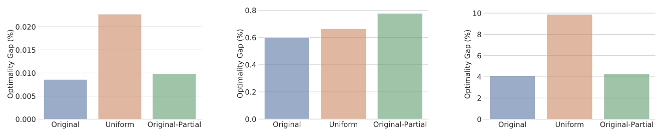


(a) Exploitability on TSP20 (b) Exploitability on TSP50 (c) Exploitability on TSP100



(d) Opt. gap of combining solvers trained on TSP20 (e) Opt. gap of combining solvers trained on TSP50 (f) Opt. gap of combining solvers trained on TSP100

Figure 2: Exploitability and performance of our model as the PSRO training goes on



(a) Opt. gap on n=20 (b) Opt. gap on n=50 (c) Opt. gap on n=100

Figure 3: Optimality gap of mixing-solver with different combined numbers and weights

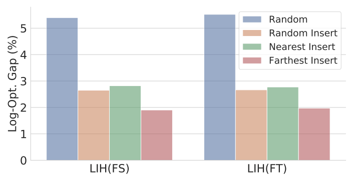


Figure 4: Ablations on different Initialization method

Fig. 3 shows the comparison between different cases. We can see 'Original' setting achieves the best results among different scales of TSP which shows the theoretic stability of nash weights described in Section. 4.4. But for 'Uniform' setting, its performance degenerates on different scales because it assigns equal importance on all solvers even though some of them are quite weak. As for the 'Original-Partial' setting, it only use 2 solvers which violates the original game structure, leading to the poor ability to deal with unseen problems. However, considering the resource consumption, we have to use partial solvers and lose a bit performance at the moment. And pursuing more efficient combining implementation and more reasonable mixing weights are left for future works.

6.3 Ablations on the initial solution

The base model LIH (Wu et al., 2021) makes improvements based on the given feasible solution so we speculate that the initial solution may have potential effects on the final solution. We here consider five initialization scenarios: random, farthest Insertion, random insertion, nearest insertion. For clear demonstration in Fig. 4, we illustrate the logarithmic mean optimal gap on TSP instances reported in Table 2. Results show that better initialization methods will lead to superior final results, which motivates us to explore better solution based on known results. Detailed information about this ablation can be seen in Apppendix. A.8.

7 Conclusions and future work

We propose a game-theoretic view on improving generalization ability of given solvers under the framework of PSRO and we are first to investigate it in this brand new perspective as far as we know. Specifically, we show the solver trained on this framework has a brilliant performance both on random generated and real world instances for TSP. We also empirically show that the consistency between the exploitability of solver's performance which can express the validity of the employment of PSRO. Various ablation studies show that there is a huge potential on the study of efficient implementation and optimal mixing weights during combining the solvers and we believe the game-theoretic training framework will have an enormous effect on solving large-scale combinatorial optimization problems.

REFERENCES

David Balduzzi, Marta Garnelo, Yoram Bachrach, Wojciech Czarnecki, Julien Perolat, Max Jaderberg, and Thore Graepel. Open-ended learning in symmetric zero-sum games. In *International Conference on Machine Learning*, pp. 434–443. PMLR, 2019.

Irwan Bello, Hieu Pham, Quoc V Le, Mohammad Norouzi, and Samy Bengio. Neural combinatorial optimization with reinforcement learning. *arXiv preprint arXiv:1611.09940*, 2016.

Yoshua Bengio, Andrea Lodi, and Antoine Prouvost. Machine learning for combinatorial optimization: a methodological tour d'horizon. *European Journal of Operational Research*, 2020.

Hanjun Dai, Elias B Khalil, Yuyu Zhang, Bistra Dilkina, and Le Song. Learning combinatorial optimization algorithms over graphs. *arXiv preprint arXiv:1704.01665*, 2017.

Constantinos Daskalakis, Paul W Goldberg, and Christos H Papadimitriou. The complexity of computing a nash equilibrium. *SIAM Journal on Computing*, 39(1):195–259, 2009.

Trevor Davis, Neil Burch, and Michael Bowling. Using response functions to measure strategy strength. In *Twenty-Eighth AAAI Conference on Artificial Intelligence*, 2014.

Xidong Feng, Oliver Slumbers, Yaodong Yang, Ziyu Wan, Bo Liu, Stephen McAleer, Ying Wen, and Jun Wang. Discovering multi-agent auto-curricula in two-player zero-sum games. *arXiv preprint arXiv:2106.02745*, 2021.

Zhang-Hua Fu, Kai-Bin Qiu, and Hongyuan Zha. Generalize a small pre-trained model to arbitrarily large tsp instances. *arXiv preprint arXiv:2012.10658*, 2020.

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

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

Wouter Kool, Herke Van Hoof, and Max Welling. Attention, learn to solve routing problems! *arXiv preprint arXiv:1803.08475*, 2018.

Wouter Kool, Herke van Hoof, Joaquim Gromicho, and Max Welling. Deep policy dynamic programming for vehicle routing problems. *arXiv preprint arXiv:2102.11756*, 2021.

Marc Lanctot, Vinicius Zambaldi, Audrunas Gruslys, Angeliki Lazaridou, Karl Tuyls, Julien Pérolat, David Silver, and Thore Graepel. A unified game-theoretic approach to multiagent reinforcement learning. *arXiv preprint arXiv:1711.00832*, 2017.

Hao Lu, Xingwen Zhang, and Shuang Yang. A learning-based iterative method for solving vehicle routing problems. In *International Conference on Learning Representations*, 2019.

Stephen McAleer, John Lanier, Roy Fox, and Pierre Baldi. Pipeline psro: A scalable approach for finding approximate nash equilibria in large games. *arXiv preprint arXiv:2006.08555*, 2020.

H Brendan McMahan, Geoffrey J Gordon, and Avrim Blum. Planning in the presence of cost functions controlled by an adversary. In *Proceedings of the 20th International Conference on Machine Learning (ICML-03)*, pp. 536–543, 2003.

Paul Muller, Shayegan Omidshafiei, Mark Rowland, Karl Tuyls, Julien Perolat, Siqi Liu, Daniel Hennes, Luke Marris, Marc Lanctot, Edward Hughes, et al. A generalized training approach for multiagent learning. *arXiv preprint arXiv:1909.12823*, 2019.

Nicolas Perez Nieves, Yaodong Yang, Oliver Slumbers, David Henry Mguni, and Jun Wang. Modelling behavioural diversity for learning in open-ended games. *arXiv preprint arXiv:2103.07927*, 2021.

Shayegan Omidshafiei, Christos Papadimitriou, Georgios Piliouras, Karl Tuyls, Mark Rowland, Jean-Baptiste Lespiau, Wojciech M Czarnecki, Marc Lanctot, Julien Perolat, and Remi Munos. α-rank: Multi-agent evaluation by evolution. *Scientific reports*, 9(1):1–29, 2019.

Gerhard Reinelt. Tsplib—a traveling salesman problem library. *ORSA journal on computing*, 3(4): 376–384, 1991.

Max Olan Smith, Thomas Anthony, Yongzhao Wang, and Michael P Wellman. Learning to play against any mixture of opponents. *arXiv preprint arXiv:2009.14180*, 2020.

Max Olan Smith, Thomas Anthony, and Michael P Wellman. Iterative empirical game solving via single policy best response. *arXiv preprint arXiv:2106.01901*, 2021.

Peter D Taylor and Leo B Jonker. Evolutionary stable strategies and game dynamics. *Mathematical biosciences*, 40(1-2):145–156, 1978.

Karl Tuyls, Julien Perolat, Marc Lanctot, Joel Z Leibo, and Thore Graepel. A generalised method for empirical game theoretic analysis. *arXiv preprint arXiv:1803.06376*, 2018.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *arXiv preprint arXiv:1706.03762*, 2017.

Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. Graph attention networks. *arXiv preprint arXiv:1710.10903*, 2017.

Oriol Vinyals, Meire Fortunato, and Navdeep Jaitly. Pointer networks. *arXiv preprint arXiv:1506.03134*, 2015.

Michael P Wellman. Methods for empirical game-theoretic analysis. In *AAAI*, pp. 1552–1556, 2006.

David H Wolpert and William G Macready. No free lunch theorems for optimization. *IEEE transactions on evolutionary computation*, 1(1):67–82, 1997.

Yaoxin Wu, Wen Song, Zhiguang Cao, Jie Zhang, and Andrew Lim. Learning improvement heuristics for solving routing problems.. *IEEE Transactions on Neural Networks and Learning Systems*, 2021.

Yaodong Yang, Rasul Tutunov, Phu Sakulwongtana, and Haitham Bou Ammar. α^α-rank: Practically scaling α-rank through stochastic optimisation. *arXiv preprint arXiv:1909.11628*, 2019.

A Appendix

A.1 Oracle Training

In the Algorithm 1, we need train two oracles: $S^{'}$ and $\mathbf{P}_{\mathcal{I}}^{'}$ as a new policy to be added to the corresponding policy set. Here we will provide a specific derivation for training the oracle in our combinatorial optimization problems setting.

Taken the formula from Eq. 6, the gradient is apparent to get:

$$\begin{aligned}
\nabla_\theta L_{\text{SS}}(\theta) &= \nabla_\theta \mathbf{E}_{\mathbf{P}_{\mathcal{I}}\sim\sigma_{\text{DG}}} \mathbf{E}_{\mathcal{I}\sim\mathbf{P}_{\mathcal{I}}} g(S_\theta, \mathcal{I}, \text{Oracle}) \\
&= \mathbf{E}_{\mathbf{P}_{\mathcal{I}}\sim\sigma_{\text{DG}}} \mathbf{E}_{\mathcal{I}\sim\mathbf{P}_{\mathcal{I}}} \nabla_\theta g(S_\theta, \mathcal{I}, \text{Oracle}) \\
&= \mathbf{E}_{\mathbf{P}_{\mathcal{I}}\sim\sigma_{\text{DG}}} \mathbf{E}_{\mathcal{I}\sim\mathbf{P}_{\mathcal{I}}} \frac{\nabla_\theta S_\theta(\mathcal{I})}{\text{Oracle}(\mathcal{I})} \\
&= \mathbf{E}_{\mathbf{P}_{\mathcal{I}}\sim\sigma_{\text{DG}}} \mathbf{E}_{N\sim\mathbf{P}_N} \mathbf{E}_{x_1,\dots,x_N\sim\prod_{i=1}^N \mathbf{P}_C} \frac{\nabla_\theta S_\theta(x_1,\dots,x_N)}{\text{Oracle}(x_1,\dots,x_N)}.
\end{aligned} \tag{15}$$

Also for Eq. 11, the computation of this gradient is:

$$\begin{aligned}
\nabla_\gamma L_{\text{DG}}(\gamma) &= \mathbf{E}_{S\sim\sigma_{\text{SS}}} \nabla_\gamma \mathbf{E}_{\mathcal{I}\sim\mathbf{P}_{\mathcal{I},\gamma}} g(S, \mathcal{I}, \text{Oracle}) \\
&= \mathbf{E}_{S\sim\sigma_{\text{SS}}} \int_{\mathcal{I}} \nabla_\gamma \mathbf{P}_{\mathcal{I},\gamma}(\mathcal{I}) g(S, \mathcal{I}, \text{Oracle}) \mathbf{d}\mathcal{I} \\
&= \mathbf{E}_{S\sim\sigma_{\text{SS}}} \int_{\mathcal{I}} \mathbf{P}_{\mathcal{I},\gamma}(\mathcal{I}) \frac{\nabla_\gamma \mathbf{P}_{\mathcal{I},\gamma}(\mathcal{I})}{\mathbf{P}_{\mathcal{I},\gamma}(\mathcal{I})} g(S, \mathcal{I}, \text{Oracle}) \mathbf{d}\mathcal{I} \\
&= \mathbf{E}_{S\sim\sigma_{\text{SS}}} \mathbf{E}_{\mathcal{I}\sim\mathbf{P}_{\mathcal{I},\gamma}} \nabla_\gamma \log \mathbf{P}_{\mathcal{I},\gamma}(\mathcal{I}) g(S, \mathcal{I}, \text{Oracle}).
\end{aligned} \tag{16}$$

Furthermore, we can take a expansion on $\mathbf{E}_{\mathcal{I}\sim\mathbf{P}_{\mathcal{I},\gamma}} \nabla_\gamma \log \mathbf{P}_{\mathcal{I},\gamma}(\mathcal{I})$ in the last line w.r.t. $\gamma = (\gamma_C, \gamma_N)$:

$$\mathbf{E}_{\mathcal{I}\sim\mathbf{P}_{\mathcal{I},\gamma}} \nabla_\gamma \log \mathbf{P}_{\mathcal{I},\gamma}(\mathcal{I}) = \begin{pmatrix} \mathbf{E}_{N\sim\mathbf{P}_{N,\gamma_N}} \mathbf{E}_{x_1,..,x_N\sim\prod_{i=1}^N \mathbf{P}_{C,\gamma_C}} \nabla_{\gamma_C} (\sum_{i=1}^N \log \mathbf{P}_{C,\gamma_C}(x_i)) \\ \mathbf{E}_{N\sim\mathbf{P}_{N,\gamma_N}} \nabla_{\gamma_N} \mathbf{E}_{x_1,..,x_N\sim\prod_{i=1}^N \mathbf{P}_{C,\gamma_C}} \log \mathbf{P}_{N,\gamma_N}(N) \end{pmatrix}$$

After taking the above formula into Eq. 16, we complete the derivation of gradients about the data generator.

A.2 Computation of Log-Probability

An extra computation is needed for the log-probability in Eq. 9 and we do this in the following way: Assuming it's independent between each dimension in a two-dimension coordinate, we only show the one-dimension case without loss of generality.

Formally, there are two random variables $X \sim \text{U}(0,1)$ and $Y \sim \text{N}(0,\sigma^2)$, and we are to compute the probability density function of random variable $Z = X + Y$. We get:

$$\text{P}(Z \le z) = \text{P}(X + Y \le z) = \int_0^1 dx \int_{-\infty}^{z-x} \frac{1}{\sqrt{2\pi}\sigma} \exp(-\frac{y^2}{2\sigma^2}) dy$$

and we have:

$$\text{p}(z) = \frac{d\text{P}(Z \le z)}{dz} = \int_0^1 \frac{1}{\sqrt{2\pi}\sigma} \exp(-\frac{(z-x)^2}{2\sigma^2}) \text{d}x.$$

All we need to do is to approximate this integration. Various methods can be used do so. In this work, we handle this simple integration by Monte Carlo sampling by sampling 10000 samples within $[0,1]$ to make a rough approximation. After obtaining the approximated probability, we can easily get the log-probability due to the independent assumption.

A.3 Detailed Experimental Settings

Hyperparameters. We don't propose any specific RL solver in this work since our method is a unified framework to suit any previous models. So in this paper, we use **LIH** as our base model. All settings about the RL solver is same as the original paper.

Table 3: Configuration of Attack Neural Networks

Module	DESCRIPTION
First Layer	dim=2 with ReLU activation
Second Layer	dim=128 with ReLU activation
Output Layer	dim=2 with Sigmoid activation
Optimizer	Adam with lr=0.05, lr_decay=0.95
l_2 norm	weight_decay=0.01
Epochs	50 for TSP20, 50 for TSP50 and TSP100

For the settings of data generator, we initialize the γ_N randomly and use a simple three-layer neural networks to represent the attack generator f_{γ_C} in Eq. 8. We also use a Sigmoid function as the last layer to scale the variation within $[0, 1]$ and an additional scalar $\lambda \in [0, 1]$ to make a further limit within $[0, \lambda]$. Here we set $\lambda = \frac{1}{3}$ because of the '68-95-99.7 rule' which is a famous principle in statistics. It not only guarantees each point within [0,1] can reach any other point after adding a gaussian perturbation, but also makes few changes to the structure of original instances after normalization in Eq. 14.

All parameters in our framework except for those in the RL solver are updated by Adam (Kingma & Ba, 2014) optimizer with specific learning rate settings and the overall configuration of this neural networks is shown in Table 3.

During the training at each PSRO loop, we choose the attack generator where the current solver selector performs worst for the next PSRO loop. Similarly, we choose the solver with best performance on the mixing distribution constructed by the current data generator. Specifically, we generate a validation set by sampling 1000 instances from the distribution constructed by the data generator's policy set and its meta-strategy. We then test each epoch's model on this dataset and select the best one as the model to be trained in the next PSRO loop.

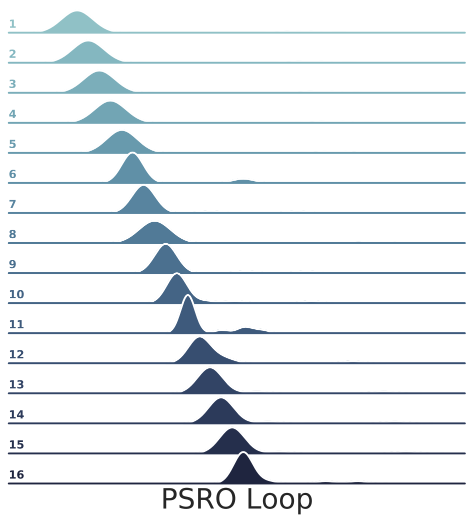
Fine-tune version. We also provide a fine-tune version under our training framework. Specifically, we pick the model as a warm start which has pretrained on the uniform distribution and continue to train them under the framework of PSRO, which can be seen as the fine-tune process to overcome the weakness of the current model. We call this version of model **LIH(FT)** in the following. Respectively, we denote the version of model that trains from scratch as **LIH(FS)**. For practical use, we often use the fine-tune model rather than the model train from scratch because of the limit of time and computational resource, and in this work, we test the fine-tune model trained on TSP100 to solve the instances from TSPLIB (Reinelt, 1991)).

Setup in meta-level. Under the framework of PSRO, we train oracles for solver selector and data generator at each PSRO loop. During training **LIH(FS)**, we set the same training epochs for RL solver: 40 epochs (per PSRO loop) for TSP20, TSP50 and TSP100 and we train 7 PSRO loop in each case.. When we train the fine-tune version, **LIH(FT)**, we use the model in the last epoch of training period in original paper as our pretrained model (for (Wu et al., 2021), we use the model trained after 200 epochs.) Then we train 10 epochs for TSP100 in each PSRO loop and train 8 PSRO loop. During the training, the solver inherit the parameters of last PSRO loop and continue to train in the new PSRO loop. Noticing that we obtain a population of solvers by 280 epochs of training, we train 280 epochs for **LIH** rather than 200 epochs in its original paper to guarantee the fair comparison.

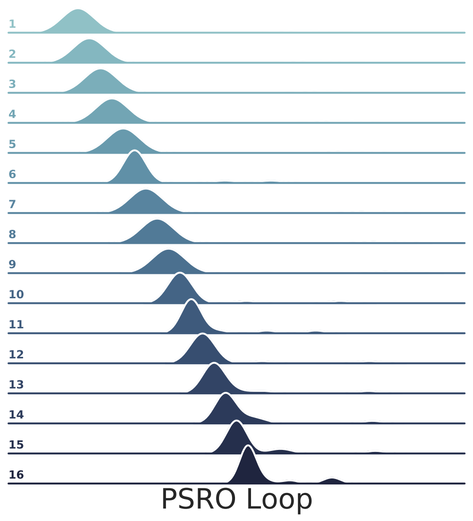
Mixing-model. After getting a population of solvers, we use the mixing policy obtained by Eq. 12 or 13 to combine these solver. Considering we need to get each solver's policy during each decision step, we need to execute forward propagation for each solver so the running time will grow linearly if there are no implementation-level tricks. As a consequence, we only use the solvers whose probabilities are accumulated no less than 0.99 because of solver selector's sparse meta-strategy.

A.4 Meta Strategy in Different PSRO Loops

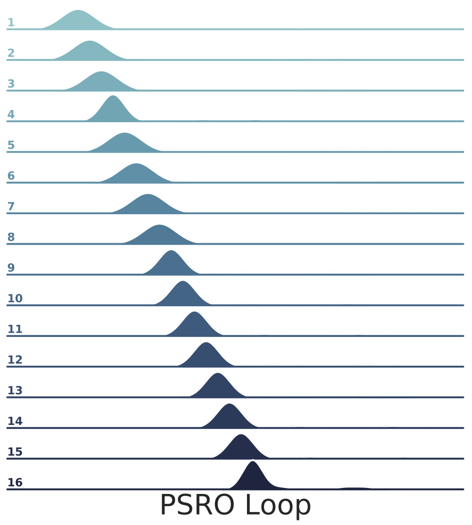
We visualize the meta strategy in every PSRO loops in Fig. 5. Results show that at each loop, there exists the strongest solver with a dominate meta strategy probability, leading to a quite sparse meta strategy distribution.



(a) Meta-strategy of the population of solvers trained on TSP20.



(b) Meta-strategy of the population of solvers trained on TSP50.



(c) Meta-strategy of the population of solvers trained on TSP100.

Figure 5: Meta strategy of the model population.

A.5 Weakness of solvers

Under our framework, it's interesting to find some distributions where the solvers (or methods) can not perform well, in some sense, which can reveal the 'weakness' of them. It can also provide a rough judgement on the stability of a method. We are amazed to find that only using simple multi-layer neural networks, the same as that during training oracles for data generator, the methods show diverse performance, as shows in Appendix. 6 and 7. Therefore, it's reasonable to take this criterion into consideration when comparing different methods. However, there are few researches about the exploration about the weakness but we think it's quite important especially in realistic applications.

We demonstrate performance can be influenced a lot even by adding small gaussian perturbations generated according to A.4 in Fig. 6 and 7. We use the model trained in corresponding paper: training 200 epochs for LIH (Wu et al., 2021) and 100 epochs for AM (Kool et al., 2018) on TSP20, TSP50 and TSP100. Results show that our attack generator can learn a distribution where the well-trained model performs bad, which motivates us to employ such method to train oracles under the framework of PSRO.

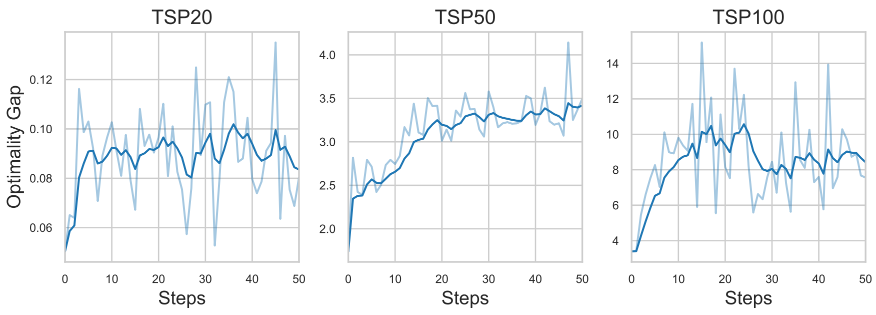


Figure 6: Training figure of attack generator for LIH (Wu et al., 2021).

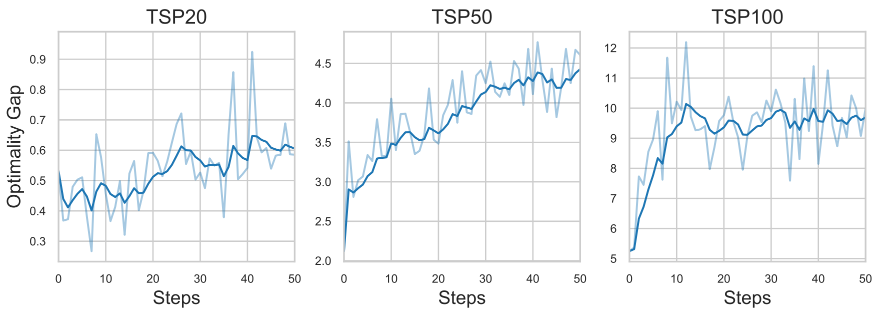


Figure 7: Training figure of attack generator for AM (Kool et al., 2018)

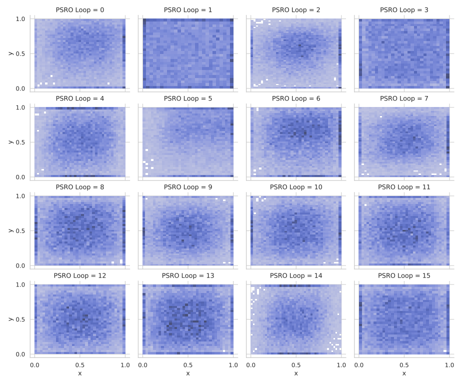
A.6 Demonstration of Attack Distribution

We visualize the attack distribution obtained by each PSRO loop in Fig. 8. Specifically, Fig. 8(a), 8(c) and 8(e) are points which comprises 1000 instances. Fig. 8(b), 8(d) and 8(f) are corresponding kernel density estimations.

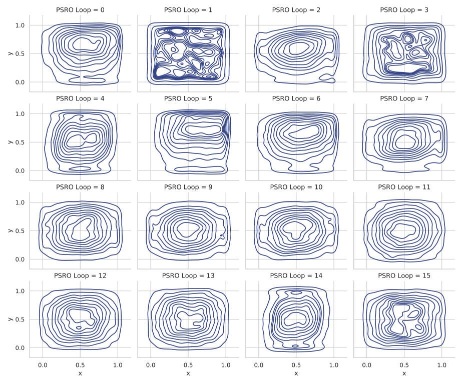
A.7 Generalization on different scales

We test the generalization ability of our trained model compared with some other baselines and results are showed in Fig. 9. As noticed in (Kool et al., 2018), the truth of No Free Lunch theorem (Wolpert & Macready, 1997) can be explained the degradation of performance on different scales. And the model can have excellent performance only when the scales of training data and testing data are same. As shown in Fig. 9(a), AM (Kool et al., 2018) represents the same generalization performance as that in its original paper. However, in Fig. 9(b) - 9(d), the generalization ability of LIH (Wu et al., 2021) seems unsatisfactory compared to AM (Kool et al., 2018).

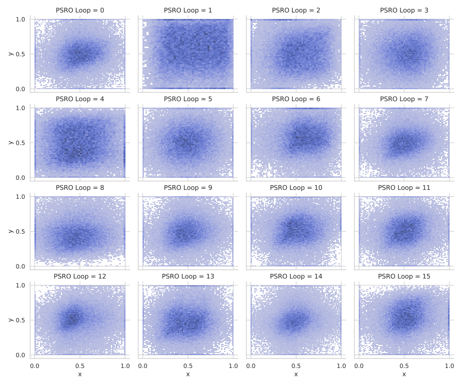
For a fair comparison between LIH in original paper and that trained under PSRO, we use the LIH trained after 280 epochs (same epoch number as ours). An interesting finding is that the generalization ability on problem scales all has an improvement by a big margin according to Fig. 9(b), 9(c), 9(d), even though we only consider the effects on distribution.



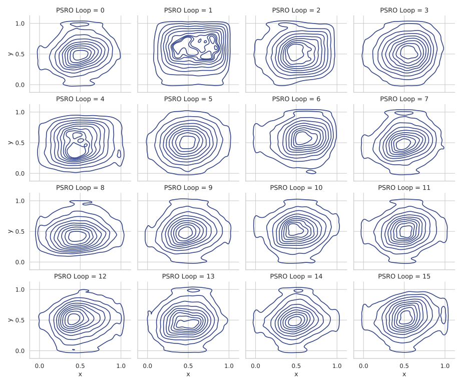
(a) Attack distribution generated by PSRO trained on TSP20.



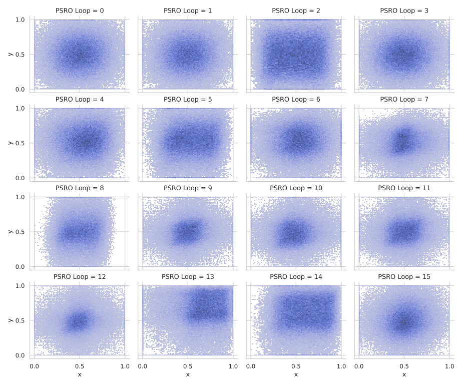
(b) Kernel density estimations of attack distribution generated by PSRO trained on TSP20.



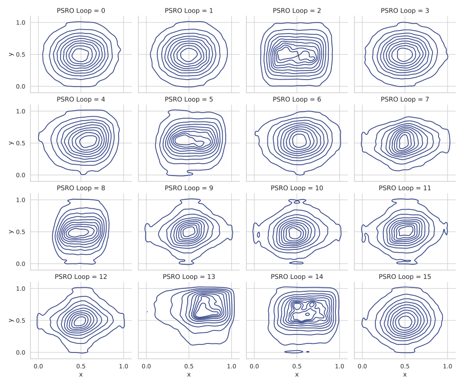
(c) Attack distribution generated by PSRO trained on TSP50.



(d) Kernel density estimations of attack distribution generated by PSRO trained on TSP50.

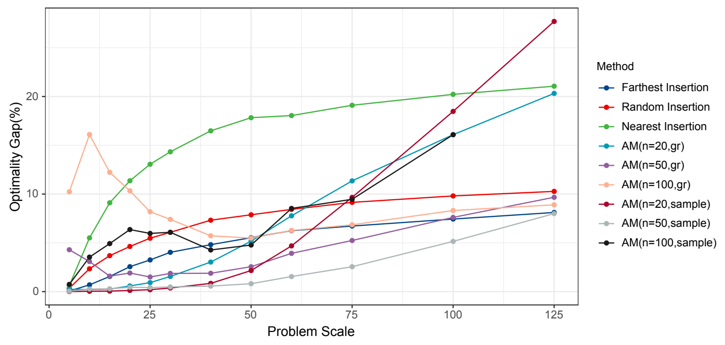


(e) Attack distribution generated by PSRO trained on TSP100.

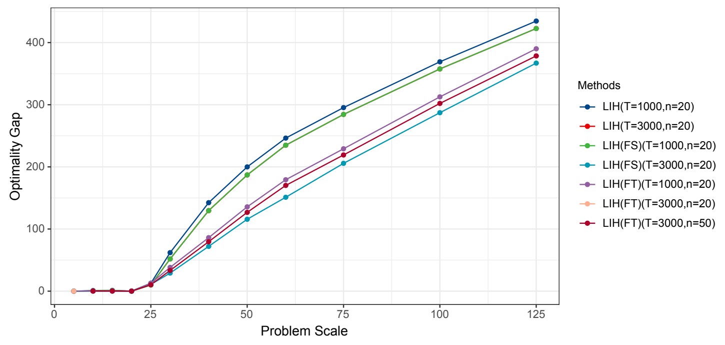


(f) Kernel density estimations of attack distribution generated by PSRO trained on TSP100.

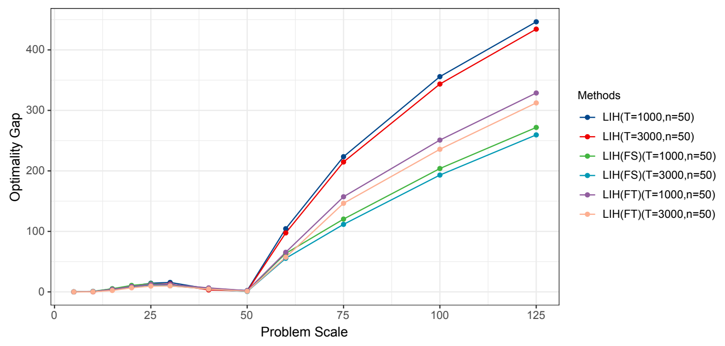
Figure 8: Attack distribution generated by PSRO



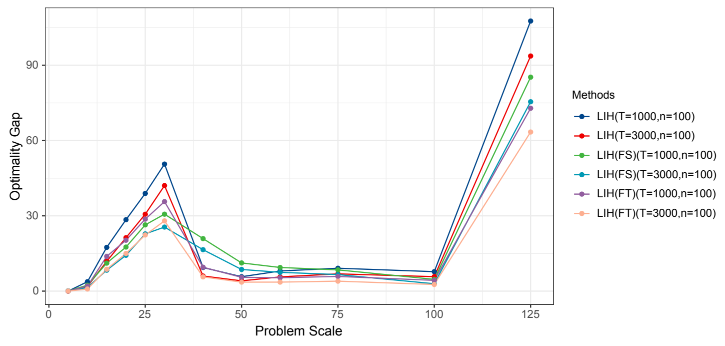
(a) Generalization results on some classical heuristic methods and AM (Kool et al., 2018).



(b) Generalization results on LIH (Wu et al., 2021) and the corresponding model trained under PSRO with n=20.



(c) Generalization results on LIH (Wu et al., 2021) and the corresponding model trained under PSRO with n=50.



(d) Generalization results on LIH (Wu et al., 2021) and the corresponding model trained under PSRO with n=100.

Figure 9: Generalization results of some models (or methods) on different problem scales.

Table 4: Ablation Results under Different Initial Solutions

Instance	Opt.	LIH(FS)				LIH(FT)			
		Random	Random Insert	Nearest Insert	Farthest Insert	Random	Random Insert	Nearest Insert	Farthest Insert
pr226	80,369	697,738	103,441	97,357	84,088	853,580	103,441	97,357	84,088
ts225	126,643	781,083	157,731	155,603	136,699	981,680	157,731	155,603	136,699
kroD100	21,294	22,717	23,288	23,377	22,346	23,624	23,139	23,377	22,346
eil51	426	451	459	437	444	462	441	437	444
kroA100	21,282	23,033	21,458	22,269	22,891	22,596	22,289	22,269	22,891
pr264	49,135	442,632	58,343	65,878	55,312	523,232	58,343	65,878	55,312
pr152	73,682	300,897	91,335	86,914	76,389	302,977	91,335	86,914	76,389
gil262	2,378	14,033	2,615	2,914	2,638	17,270	2,615	2,914	2,638
rat99	1,211	1,298	1,299	1,305	1,250	1,325	1,341	1,305	1,250
kroA150	26,524	97,431	28,628	31,344	28,789	86,237	28,628	31,344	28,789
lin105	14,379	21,526	18,102	18,170	15,372	19,102	17,758	18,170	15,372
pr124	59,030	151,075	67,163	68,178	61,645	160,136	67,163	68,178	61,645
st70	675	742	702	782	741	735	699	782	741
a280	2,579	16,878	3,084	2,987	3,018	19,777	3,084	2,987	3,018
rd100	7,910	8,852	8,580	8,620	8,180	8,608	8,456	8,620	8,180
pr136	96,772	265,334	131,880	106,059	104,429	239,330	131,880	106,059	104,429
pr76	108,159	111,646	111,712	119,838	109,174	116,986	120,441	119,838	109,174
kroA200	29,368	150,172	31,824	36,029	31,450	163,725	31,824	36,029	31,450
kroB200	29,437	142,464	32,923	36,532	31,656	151,026	32,923	36,532	31,656
pr107	44,303	73,698	51,793	53,127	45,288	70,374	51,793	53,127	45,288
kroB150	26,130	84,105	27,418	31,619	28,156	83,013	27,418	31,619	28,156
u159	42,080	173,829	52,854	52,005	46,771	158,815	52,854	52,005	46,771
berlin52	7,542	9,200	7,845	8,379	7,758	8,185	7,775	8,379	7,758
rat195	2,323	8,716	3,079	2,953	2,600	11,645	3,079	2,953	2,600
d198	15,780	56,201	17,759	17,839	16,501	71,529	17,759	17,839	16,501
eil101	629	659	659	657	641	686	664	657	641
pr144	58,537	275,619	84,936	73,306	62,522	245,762	84,936	73,306	62,522
pr299	48,191	368,945	61,035	59,699	51,726	380,381	61,035	59,699	51,726
kroC100	20,749	21,957	21,413	23,144	21,147	22,766	21,589	23,144	21,147
tsp225	3,916	19,333	4,514	4,774	4,262	19,814	4,514	4,774	4,262
eil76	538	566	568	590	569	559	565	590	569
kroB100	22,141	23,413	23,459	23,414	23,927	25,763	23,895	23,414	23,927
kroE100	22,068	23,563	23,750	24,135	23,172	24,257	23,589	24,135	23,172
ch150	6,528	20,633	7,145	7,917	6,866	25,626	7,145	7,917	6,866
bier127	118,282	180,026	132,501	144,569	127,520	189,875	132,501	144,569	127,520
ch130	6,110	15,097	6,495	7,436	6,655	15,884	6,495	7,436	6,655
Avg. Gap (%)	0	221.18	14.13,	16.77	6.70	250.24	14.35	16.00	7.19

A.8 Results on Different Initialization Method

Based on the mechanism of LIH (Wu et al., 2021), we can make improvements based on given feasible solution. In this part, we investigate the performance of our model under different initial solutions on TSPLib instances. Specifically, we consider four scenarios:

- Random: Generating random solution for the given instance
- Random Insert: Generating initial solution using random insertion heuristic
- Nearest Insert: Generating initial solution using Nearest insertion heuristic
- Farthest Insert: Generating initial solution using Farthest insertion heuristic

Table. 4 shows that with Farthest Insertion heuristic, our model can attain excellent performances on these real world instances compared with other initial method, especially with random generation. Recall the results in Table. 1, Farthest Insertion heuristic has better performance than Random and Nearest Insertion heuristic, which motivates us to treat the deep-learning based methods as heuristics to improve known best results which may be not optimal.