ECAI 2024
U. Endriss et al. (Eds.)

doi:10.3233/FAIA240707

Take a Step and Reconsider: Sequence Decoding for Self-Improved Neural Combinatorial Optimization

Jonathan Pirnay[a,b] **and Dominik G. Grimm**[a,b,*]

[a]Technical University of Munich, TUM Campus Straubing
[b]University of Applied Sciences Weihenstephan-Triesdorf

Abstract. The constructive approach within Neural Combinatorial Optimization (NCO) treats a combinatorial optimization problem as a finite Markov decision process, where solutions are built incrementally through a sequence of decisions guided by a neural policy network. To train the policy, recent research is shifting toward a 'self-improved' learning methodology that addresses the limitations of reinforcement learning and supervised approaches. Here, the policy is iteratively trained in a supervised manner, with solutions derived from the current policy serving as pseudo-labels. The way these solutions are obtained from the policy determines the quality of the pseudo-labels. In this paper, we present a simple and problem-independent sequence decoding method for self-improved learning based on sampling sequences without replacement. We incrementally follow the best solution found and repeat the sampling process from intermediate partial solutions. By modifying the policy to ignore previously sampled sequences, we force it to consider only unseen alternatives, thereby increasing solution diversity. Experimental results for the Traveling Salesman and Capacitated Vehicle Routing Problem demonstrate its strong performance. Furthermore, our method outperforms previous NCO approaches on the Job Shop Scheduling Problem.

1 Introduction

Combinatorial optimization (CO) problems, which are characterized by their discrete nature and often NP-hard complexity, are essential in many areas, including logistics, manufacturing, process design, and scheduling. Traditional methods often rely on heuristics that require domain expertise and struggle with scalability. In recent years, Neural Combinatorial Optimization (NCO) has emerged as a research area that aims to let deep neural networks learn to generate heuristics from the instance distribution of the problem at hand [4]. Among various strategies within NCO, the *constructive* approach formulates a CO problem as a finite Markov decision process and represents a solution to an instance as a *sequence* of incremental decisions. A neural network computes a policy to guide these decisions.

The policy network is typically trained with supervised learning (SL) techniques [38, 14, 9, 21, 8, 24] or reinforcement learning (RL) [3, 17, 27, 12, 22, 15, 28, 40, 29, 42, 13, 43]. Both approaches have particular challenges: SL-based methods require a large number of high-quality expert solutions to be used as labels, which are usually obtained from existing (exact) solvers. In the case of larger instances or complex problems, it can be challenging, if not infeasible, to pregenerate high-quality solutions. Conversely, RL-based methods do not necessitate the use of expert solutions, yet they are susceptible to the sparse reward problem [10] and high hyperparameter sensitivity [32]. Furthermore, RL-based approaches typically use policy gradient methods (in particular, variants of REINFORCE [39]), where gradients are derived from complete trajectories, resulting in high computational cost. While state-of-the-art constructive RL-based approaches such as POMO [22] demonstrate remarkable performance on the training distribution, they exhibit limited generalizability to larger problem instances. To further complicate matters, it has recently been shown [24, 8] that the poor generalization is likely due to the lightweight decoder structure of commonly used architectures [17]. Instead, Luo et al. [24] and Drakulic et al. [8] suggest increasing the decoder size, which leads to significant memory requirements for policy gradient methods.

To bridge SL- and RL-based methods, recent work [24, 6, 30, 25] diverges from RL to a more straightforward, *self-improved learning* (SIL) approach: During training, the current policy network is decoded (e.g., by sampling) to generate (or refine) solutions to randomly generated instances. The best solutions generated are used as pseudo-labels, which the network is trained to imitate via SL. Repeating this process creates a "self-improving" loop. As no gradients need to be collected during decoding, large architectures can be trained in this way. The challenge lies in identifying an effective decoding method that is **(a)** capable of rapidly generating solutions for potentially thousands of instances that can be utilized in a single training epoch; **(b)** able to provide diverse sequences for sufficient exploration but can be adapted to exploit the model in later training stages; and, **(c)** generalizable across different problems, with few hyperparameters to adjust to the specific problem at hand. Given **(a)**, time-consuming search methods such as Monte Carlo Tree Search (MCTS) are unsuitable. Although naive Monte Carlo i.i.d. sampling from the policy is theoretically sound [6], its output is often not diverse. Furthermore, it can require a large number of samples (even at low annealing temperatures or Top-p/-k sampling [11]) to let the model improve in later stages of training [19, 33, 22, 30]. On the other hand, sampling sequences *without replacement* (WOR) [19, 33, 26] yields diverse sequences. However, for CO problems, it has been observed that the advantage of sampling WOR over sampling with replacement diminishes with increasing solution length [33, 30].

In this paper, we propose a simple yet effective decoding mechanism for sequence models that exploits the diversity and paralleliza-

* Corresponding Author. Email: dominik.grimm@{hswt,tum}.de
Code available at: https://github.com/grimmlab/step-and-reconsider

tion capabilities of Stochastic Beam Search (SBS) [19] in an MCTS-like manner: We maintain a search tree where each node represents a partial solution and leaf nodes are complete solutions. Given a beam width k and a step size s, we sample k leaf nodes WOR from the model using SBS. We then remove the probability mass of the leaves from the tree, which marks the sequences as sampled. We select the best solution from the sampled leaves and, assuming it corresponds to a sequence $(a_1, \dots, a_n)$, follow it for s steps. After shifting the tree's root to the partial solution $(a_1, \dots, a_s)$, we repeat the process of finding a better solution until we have traversed the entire tree.

The decoding method is fast, generalizable, and possesses only two intuitive hyperparameters, namely k and s. These can be readily adapted to the available computational resources and problem length. By marking found sequences as sampled, we consistently consider *unseen* alternative solutions. By following the best solution for s steps, we gradually reduce the length of the problem. This forces more diversity in the sampling process.

We summarize our contributions as follows:

- In the recent spirit of simplifying and scaling the training process of NCO methods, we propose a novel and straightforward sequence decoding method for SIL.
- We train two state-of-the-art architectures [8, 24] for the Traveling Salesman Problem (TSP) and the Capacitated Vehicle Routing Problem (CVRP) with 100 nodes in an SIL setting using our decoding method. Our method matches the performance of SL on expert trajectories when evaluated on the training distribution and shows similarly strong generalization performance on larger instances.
- We further evaluate our method on the Job Shop Scheduling Problem (JSSP), consistently outperforming current state-of-the-art NCO methods.
- We additionally show on various policies that our proposed decoding method significantly outperforms SBS-based sampling with the same computational budget.

2 Related work

Constructive NCO The first application of neural networks to directly predict solutions to CO problems is attributed to the Pointer Network of Vinyals et al. [38]. Originally trained via SL, Bello et al. [3] employ REINFORCE [39] with a learned value baseline. Since then, as in many other areas of deep learning, variants of the Transformer [36] have become the standard architecture choice for many NCO models [17, 7, 22, 23, 8, 24, 44]. To circumvent learning a value function, the policy networks are usually trained with self-critical policy gradient methods [31] over complete trajectories. In particular, POMO [22] exploits problem symmetries and samples solutions from every possible starting node for a single instance, thereby significantly diversifying the solutions found. POMO and similar RL-methods perform remarkably well on training distributions of up to 100 nodes in routing problems. However, they do not scale well to larger instance sizes. The recent methods BQ [8] and LEHD [24] attribute the poor generalization to the light decoder structure of the used Attention Model [17]. In contrast, they propose significantly increasing the decoder size (e.g., up to nine transformer blocks in BQ). Drakulic et al. [8] and Luo et al. [24] train their models with SL on expert solutions to instances with only 100 nodes for routing problems and achieve state-of-the-art results when generalizing up to 1,000 nodes. However, the size of the architecture makes it challenging to train with policy gradient methods.

Self-improved learning To overcome the difficulties associated with RL and SL for NCO, recent studies propose a "self-improving" training paradigm. The central concept during training is to use the current policy and generate solutions (i.e., sequences) to random instances, which are then used as pseudo-labels to train the network with SL in a next-token prediction setup. In the appendix of their LEHD paper, Luo et al. [24] describe a self-improvement method where the model is pre-trained with RL on small routing problems to be computationally feasible. The resulting model is used to generate solutions to a set of randomly generated larger problem instances. Exploiting problem symmetries, the solutions are further improved by re-unrolling the policy along random subtours. The policy is then trained to imitate the resulting set of solutions. With impressive results, Luo et al. [25] further develop this approach for up to 100,000 nodes. However, the method is limited to routing problems where an optimal solution of a complete tour guarantees the optimality of any subtour. Corsini et al. [6] propose a "self-labeling" strategy for the JSSP. They utilize vanilla Monte Carlo i.i.d. sampling with the current model during training to obtain increasingly optimal solutions for the model to imitate. Pirnay and Grimm [30] employ a similar training strategy. They improve the sampling process by sampling solutions WOR over multiple rounds, with each round guiding the policy towards sequences that perform better than expected. This method is close to ours in the spirit of diversifying sequences by sampling WOR in multiple steps. However, their method requires scaling the advantages with problem- and training-dependent hyperparameters, which can be complex to tune.

Sequence decoding There is a plethora of search methods at *inference time* to improve on the greedy output of a sequence model besides pure sampling or beam search, e.g. [3, 13, 5, 11]. Related to our approach is Simulation-guided Beam Search [5], which performs beam search in an MCTS way by coupling the pruning step in beam search with greedy rollouts. However, due to its purely exploitative nature, it is an unsuitable choice for sequence decoding in the context of SIL. In general, AlphaZero-type algorithms [34] share similarities with SIL. However, running multiple simulations in the MCTS for a single action choice is time-consuming and non-trivial to parallelize. Concerning sampling, considering a diverse set of solutions can significantly improve training [22, 15]. A prominent way of diversification is sampling WOR [19, 33, 26]. In this work, we use SBS [19], as it can be parallelized in a manner analogous to regular beam search.

3 Preliminaries

3.1 Problem formulation

We consider a CO problem with n discrete decision variables. A solution to a problem instance is given by a tuple $(a_1, \dots, a_n)$, representing the n variable assignments (we assume a given numerical order). Let S be the space of all possible solutions $\bar{a}_{1:n} := (a_1, \dots, a_n)$. The goal is to find a solution that maximizes a predefined objective function $f\colon S \to \mathbb{R} \cup \{-\infty\}$. Here, f maps infeasible solutions to $-\infty$.

The constructive approach formulates the problem autoregressively, where a value is chosen for a_1, then for a_2 given a_1, and so on, until a full solution $\bar{a}_{1:n}$ is created. The policy network π_θ to guide these incremental choices is a sequence model with parameters θ. For a partial solution $\bar{a}_{1:d} = (a_1, \dots, a_d)$, with $d < n$, the policy π_θ computes the conditional distribution $\pi_\theta(a_{d+1}|\bar{a}_{1:d})$ over the choices a_{d+1} for the $(d+1)$th decision variable. In particular, to

obtain a complete solution from the policy, we begin with an empty tuple $\bar{a}_{1:0} := ()$ and autoregressively decode $a_d \sim \pi_\theta(\cdot|\bar{a}_{1:d-1})$. For a (partial) solution $\bar{a}_{1:d} = (a_1, \dots, a_d)$ with $d \leq n$, we denote by $\pi_\theta(\bar{a}_{1:d})$ the total probability $\pi_\theta(\bar{a}_{1:d}) = \prod_{i=1}^{d} \pi_\theta(a_i|\bar{a}_{1:i-1})$.

To simplify the notation, we do not use explicit labels to distinguish problem instances. However, it will always be transparent in context to which instance a solution is to be assigned. We assume a constant problem length n for simplicity, but our method applies equally to varying sequence lengths.

3.2 Self-improved training cycle

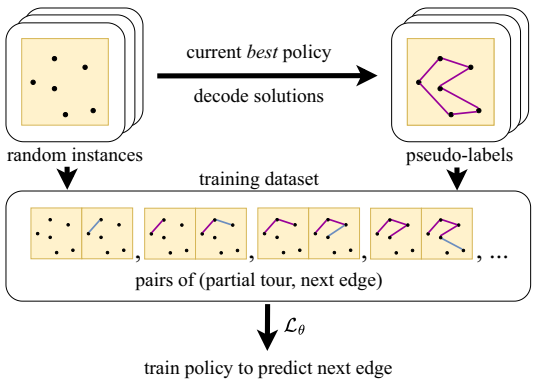

Figure 1. Overview of self-improved training with TSP as an illustrative example. A partial solution corresponds to a partial (unfinished) tour. The next sequence element for the model to predict is the next edge to be appended to the partial tour.

The model π_θ is trained using the following simple SIL strategy, which we generalize from [6, 30, 25]. We present an illustration of the training cycle in Figure 3.2. We randomly initialize parameters θ and keep best parameters θ'. Initially, we set $\theta' \leftarrow \theta$. We repeat the following steps for each training epoch:

(1) Generate a set of random problem instances. For each instance, we use $\pi_{\theta'}$ to decode multiple solutions *in some way*, and keep the best solution found, according to the objective function f. The random instances and their best solutions found (used as pseudo-labels) build the training set for this epoch.

(2) Train π_θ on the generated data to predict the next sequence element from a partial solution with a cross-entropy loss: For a batch of size B with solutions $\bar{a}^j_{1:n} = (a^j_1, \dots, a^j_n)$ for $j \in \{1, \dots, B\}$, we uniformly sample a partial solution $\bar{a}^j_{1:d_j}$ for each j and $d_j < n$. The loss to be minimized is then given by

$$\mathcal{L}_\theta = -\frac{1}{B} \sum_{j=1}^{B} \log \pi_\theta \left(a^j_{d_j+1} | \bar{a}^j_{1:d_j} \right). \tag{1}$$

(3) At the end of the epoch, evaluate π_θ and $\pi_{\theta'}$ *greedily* on a fixed validation set. If π_θ outperforms $\pi_{\theta'}$, update the best parameters $\theta' \leftarrow \theta$.

In (1), for many problems, it is desirable to generate thousands of instances in each epoch (for comparison, the models in BQ [8] and LEHD [24] are trained with SL on 1M random instances and their optimal solutions). Consequently, the efficiency of the SIL strategy is strongly determined by the method used to decode the sequence model.

4 Method

This section presents our main contribution, a sequence decoding method for efficient SIL. We also briefly recall SBS [19], which forms the backbone of our method. In the following, we omit the parameters θ in the subscript of π_θ.

4.1 Sequence decoding as tree traversal

As common for neural sequence models, we can view decoding π for a problem instance as traversing a search tree from root to leaf. The root node corresponds to an empty sequence. A node in the tree at depth d corresponds uniquely to a partial solution $\bar{a}_{1:d} = (a_1, \dots, a_d)$, and the direct children of this node represent the possible assignments to the $(d+1)$th decision variable. To be explicit, let $\mathrm{Ch}(\bar{a}_{1:d})$ be the set of direct children of $\bar{a}_{1:d}$, then any $\bar{b}_{1:d+1} = (b_1, \dots, b_{d+1}) \in \mathrm{Ch}(\bar{a}_{1:d})$ satisfies $b_i = a_i$ for $1 \leq i \leq d$. Thus, a leaf node corresponds uniquely to a complete solution.

Before explaining how we search the tree, we briefly describe how to *maintain* the tree. When decoding the model for an instance, we create a search tree in memory and expand nodes as needed. When expanding a node $\bar{a}_{1:d}$, we query the model $\pi(\cdot|\bar{a}_{1:d})$ for the transition probabilities of its children. As described later, the transition probabilities will be modified during the search. Hence, for each node $\bar{a}_{1:d}$, we additionally keep an *unnormalized* total probability $p(\bar{a}_{1:d})$ which is set to the total probability $\pi(\bar{a}_{1:d})$ when the node is created. For a node $\bar{a}_{1:d}$ with parent $\bar{a}_{1:d-1}$, we denote by $\tilde{\pi}(a_d|\bar{a}_{1:d-1})$ the *normalized* transition probability

$$\tilde{\pi}(a_d|\bar{a}_{1:d-1}) = \frac{p(\bar{a}_{1:d})}{\sum_{\bar{b}_{1:d} \in \mathrm{Ch}(\bar{a}_{1:d-1})} p(\bar{b}_{1:d})}. \tag{2}$$

4.2 Stochastic Beam Search

Ranking nodes by their total log-probability, a *beam search* of some beam width $k \in \mathbb{N}$ is a standard decoding method to obtain a set of k unique high-probability sequences from the sequence model π. In beam search, all nodes within the current beam can be evaluated by π in parallel, which aligns well with the effectiveness of GPUs on batches. Besides being classically a deterministic inference method, the k sequences found with beam search often lack diversity. Kool et al. [19] present SBS, an elegant modification of beam search to sample k sequences *without replacement* (WOR) from the sequence model π. The main idea is to perform regular beam search but perturb the total log-probability $\log \pi(\bar{a}_{1:d})$ by adding noise sampled from a standard Gumbel distribution. The Gumbel noise is sampled under the condition that the maximum perturbed log-probability of sibling nodes is equal to their parent's. This persists a node's perturbation down its subtree.

The authors show that by sampling WOR from the distribution of complete sequences, SBS can obtain a set of sequences with high diversity. As it only changes the scoring of nodes, SBS can be implemented and, importantly, parallelized as a regular deterministic beam search.

4.3 Take a step and reconsider

We can now introduce our proposed decoding method. We summarize the method in Algorithm 1, illustrate it in Figure 2, and give a brief walkthrough below.

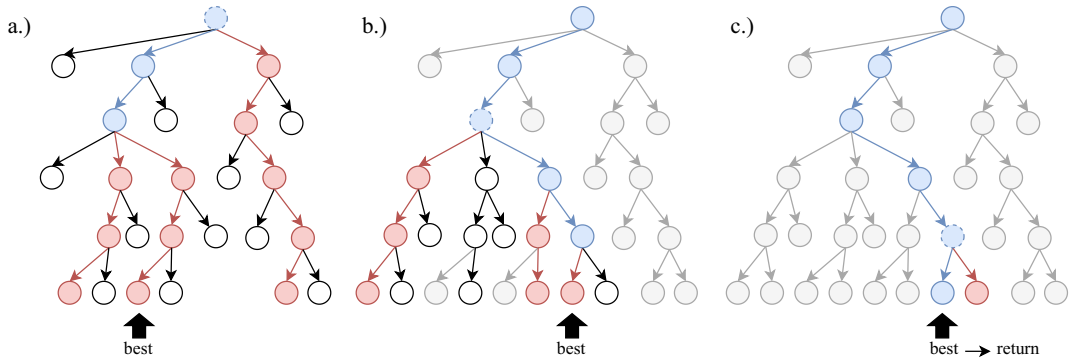

Figure 2. Example of sequence decoding with beam width $k = 3$ and step size $s = 2$. **a.)** We sample k leaves WOR (indicated in red) from the root node (dashed outline), creating nodes on demand. We follow the trajectory of the best solution for s steps (indicated in blue). **b.)** We shift the root node s, disregard the rest of the tree and remove the probability mass of sampled leaves (grayed out) from their ancestors. We sample k unseen alternatives from the new root and find a better solution. We follow the new solution for s steps. **c.)** After shifting the root again, only one leaf is left to sample, which does not improve the current best solution.

Algorithm 1: Sequence decoding for SIL in NCO

Input: $k \in \mathbb{N}$: beam width; $s \in \mathbb{N}$: step size
Input: π: policy, f: objective function for problem instance

ROOT $\leftarrow ()$ ▷ root node
$t \leftarrow 0$ ▷ step count
$\bar{b}_{1:n} \leftarrow$ NULL ▷ best solution
while $t < n$ **do**
  $L = \{\bar{a}^1_{1:n}, \dots, \bar{a}^k_{1:n}\} \leftarrow$ SBS with $\tilde{\pi}$ (see Eq. (2)) and beam width k from ROOT
  $\bar{b}_{1:n} \leftarrow \arg\max_{\bar{a}_{1:n} \in L \cup \{\bar{b}_{1:n}\}} f(\bar{a}_{1:n})$
  $t \leftarrow \min\{t + s, n\}$
  foreach $\bar{a}_{1:n} = (a_1, \dots, a_n) \in L$ **do**
    if $\bar{a}_{1:t} = \bar{b}_{1:t}$ **then**
      for $i = t, \dots, n$ **do**
        $p(\bar{a}_{1:i}) \leftarrow p(\bar{a}_{1:i}) - \pi(\bar{a}_{1:n})$ ▷ mark $\bar{a}_{1:n}$ as sampled in ancestor nodes
  ROOT $\leftarrow \bar{b}_{1:t}$
return $\bar{b}_{1:n}$

Algorithm We require the choice of two hyperparameters: a beam width $k \in \mathbb{N}$ and a step size $s \in \mathbb{N}$ with $s \leq n$. Given a problem instance, we start with an empty tree and sample a set L of k leaves $L = \{\bar{a}^1_{1:n}, \dots, \bar{a}^k_{1:n}\}$ WOR using SBS with policy $\tilde{\pi}$ (which equals π at the beginning). We take the best trajectory $\bar{b}_{1:n}$ sampled according to the objective function f. We check for every sampled sequence $\bar{a}^j_{1:n} \in L$ if it shares its first s steps with $\bar{b}_{1:n}$, i.e., if $(a^j_1, \dots, a^j_s) = (b_1, \dots, b_s)$. If so, we *mark it as sampled* by removing the leaf's probability $\pi(\bar{a}^j_{1:n})$ from its i-th ancestor's unnormalized probability mass $p(\bar{a}^j_{1:i})$, where $1 \leq i \leq n$.

Now we follow $\bar{b}_{1:n}$ for s steps and set $\bar{b}_{1:s}$ as the new root of the tree. We repeat the process above from the new root, noting that we use SBS with the *updated* sequence model $\tilde{\pi}$ (cf. (2)) from which already seen sequences can no longer be sampled. We update the currently best sequence $\bar{b}_{1:n}$ if the newly sampled k solutions contain a better one. Again, we mark the found sequences as sampled, follow the best solution for another s steps, and so on until we have traversed the entire tree.

Note that for numerical stability, we work with log-probabilities in practice.

Exploring unseen alternatives The critical thing to note is that by removing the probability mass of a leaf node from all its ancestors, sampling from the updated policy $\tilde{\pi}$ for SBS becomes equivalent to sampling WOR from π *under the condition* that already encountered sequences can not be sampled again. This strategy was originally proposed by [20, 33] to sample sequences WOR incrementally. By taking only a limited number of steps s from the best solution found and sampling again, the model is forced to consider only unseen alternatives. Furthermore, the length of the sequences to sample gradually shrinks. This is beneficial when refining the best solution further down in the tree, as sampling WOR is much more potent (compared to sampling with replacement) on shorter sequences [33, 18, 30].

Simplicity One of the advantages of SIL approaches lies in the simplicity of its training cycle (see Section 3.2), as it can foster reproducibility and adaption of the method. We aim to devise the sequence decoding - the heart of SIL - in the same spirit. The interpretation of the two hyperparameters is intuitive: k is the number of sequences to consider before temporarily committing to a solution ('the more, the better'), and s is how long to commit to a solution before exploring alternatives ('the shorter, the better'). Furthermore, k and s can be easily adjusted to available computational resources (or as training progresses), where $s = 1$ leads to an MCTS-like search, and $s = n$ reduces to simple SBS with width k.

For inference We couple the sequence decoding with Top-p (nucleus) sampling [11] to use it as an inferencing method for a trained policy. Here, the unreliable tail of the distribution is trimmed from $\tilde{\pi}$ in each expansion step of SBS. We evaluate its effectiveness in Section 5.

5 Experiments

We evaluate the efficiency of our sequence decoding method within the SIL framework (cf. Section 3.2) on the Euclidean TSP and CVRP and the standard JSSP.

Code Our code in PyTorch and models are available at https://github.com/grimmlab/step-and-reconsider. We perform training and evaluation using four NVIDIA GeForce RTX 3090 with 24GB RAM.

5.1 Routing problems

Traveling Salesman Problem A problem instance of the two-dimensional Euclidean TSP in the unit square is given by the coordinates of N nodes $x_1, \dots, x_N \in [0,1]^2 \subseteq \mathbb{R}^2$. The goal is to find a *roundtrip* that minimizes the total tour length, where the distance between two nodes x_i, x_j is given by the Euclidean norm $\|x_i - x_j\|_2$. A complete tour is constructed sequentially by choosing one unvisited node after another (see Figure 1).

Capacitated Vehicle Routing Problem In the CVRP, a *delivery vehicle* of capacity $D \in \mathbb{R}_{>0}$ needs to visit N customer nodes $x_1, \dots, x_N \in [0,1]^2 \subseteq \mathbb{R}^2$. Each customer x_i has a demand $\delta_i \in \mathbb{R}_{>0}$, which must be fulfilled by the vehicle. A feasible solution is given by a set of subtours which all start and end at a given *depot node*, where all customers are visited, and the sum of customer demands satisfied by each subtour does not exceed the capacity D. The aim is to find a feasible solution with minimal total tour length. Following the standard constructive formulation [17, 8, 24], visiting the depot is not seen as a separate step: a complete tour is constructed by deciding for each unvisited customer node whether it is reached via the depot or directly from the previous customer. This ensures solution alignment, as the length of two feasible solutions is the same, even if they contain a different number of subtours. Our problem setup is identical, so we refer to [24] for details.

Data generation and optimal solutions We consider TSP and CVRP instances of size $N \in \{100, 200, 500\}$. We generate random problem instances in the standard way by sampling node coordinates uniformly from the unit square. Additionally, for the CVRP, demands are sampled uniformly from $\{1, \dots, 9\}$. The capacity of the delivery vehicle is set to 50, 80, and 100 for a corresponding number of nodes 100, 200, and 500. For both problems, we train only on instances of size $N = 100$ and test generalization on the larger sizes. For comparison with SL, we generate a random training dataset of one million instances and a validation set of 10k instances. The test set consists of 10k instances for $N = 100$ (same set used in [17] for the TSP and [24] for the CVRP) and 128 instances for $N \in \{200, 500\}$ (same sets used in [8] for the TSP and [24] for the CVRP). For the TSP, we obtain optimal solutions from the Concorde solver [1] to use as labels in SL and compute optimality gaps. For the CVRP, (near) optimal solutions are obtained from HGS [37].

Policy network architecture We evaluate our approach using two recent state-of-the-art architectures for routing problems: the BQ architecture by Drakulic et al. [8] and the LEHD architecture by Luo et al. [24]. Both architectures are based on the Transformer [36] with a heavy decoder structure. In the original works, the models obtain strong generalization results but are trained with SL on expert data as the large architectures are unsuitable for training with RL. For CVRP with BQ, we stick to the original setup of nine transformer blocks with 12 attention heads and a latent dimension of 192. For TSP with BQ, the number of layers is the same, but with eight attention heads and a latent dimension of 128. For CVRP and TSP with LEHD, we use six transformer blocks in the decoder with eight heads and a latent dimension of 128. Similar to BQ, we use ReZero normalization [2] also for LEHD, as we found the training to be more stable (compared to no normalization as suggested in the original paper). For BQ and LEHD, the hidden dimension of the feedforward network in a transformer block is set to 512.

Training For the SIL training, we decode in each epoch solutions in parallel to 1,000 random instances. We use a beam width of $k = 64$ and step size $s = 10$. Generating the solutions takes about 2 minutes on our setup. Using the generated best solutions as pseudo-labels, we train the model on 1,000 batches of 1,024 uniformly sampled subtours as in [8]. We apply the same training structure to LEHD. We use the Adam [16] optimizer with an initial learning rate of 2e-4, clipping gradients to unit norm. To evaluate the improvement of the model, we test the policy on the pre-generated validation set after each epoch. We train the policy until we see no improvement on the validation set for 50 epochs. The setup is the same for both routing problems. However, we found the generalization performance for the CVRP to improve noticeably when, after the regular training, finetuning the model on solutions where the policy was heavily exploited. To this end, we decode another 30k solutions with $k = 256$ and $s = 1$ and Top-p sampling with $p = 0.8$ and continue to train the model for another 100 epochs. For comparison with SL, we use the same training setup but sample batches from the pre-generated training set of 1M instances.

In total, this amounts to training the BQ model with SIL for $\sim$3k epochs on the TSP ($\sim$4.5k with SL) and $\sim$3k epochs on the CVRP ($\sim$1.5k with SL). The LEHD model converges faster. With SIL, it is trained for $\sim$2k epochs on the TSP ($\sim$2k with SL) and $\sim$1k epochs on the CVRP ($\sim$1k with SL).

Baselines The primary baselines are given by SL with the corresponding identical BQ or LEHD architecture. Furthermore, we include four common constructive NCO baselines, namely **(a)** the widely used Attention Model (AM) with a beam search of width 1,024 [17], **(b)** its multi-decoder counterpart (MDAM) [41] with beam search of width 50, **(c)** POMO [22], the state-of-the-art constructive method on the training distribution, with their most potent inference technique, and **(d)** Simulation-guided Beam Search (SGBS) [5] with POMO backbone and parameters (β, γ) set to $(10, 10)$ for TSP and $(4, 4)$ for CVRP. As comparison partners for SIL methods, we list the results (TSP only) of **(e)** LEHD pre-trained with RL on small-scale instances and finetuned with SIL (LEHD RL+SIL) as reported in [24], and **(f)** the Gumbeldore (GD) training strategy (GD SIL (BQ resp. LEHD)) [30], where the sampling process is pushed toward regions with higher advantage.

Results We summarize the results in Table 1 and group them by the used architectures. Bold indicates the best optimality gap per group. Results for LEHD RL+SIL and GD SIL are taken from the original papers [24, 30]. On the TSP, we obtain excellent greedy results that even outperform the SL counterpart on the training distribution of $N = 100$, with similarly strong generalization capabilities. In particular, we outperform GD SIL, which has a more complex decoding strategy. The same dynamic can be observed on the CVRP with BQ, with worse but still strong generalization results. Our SIL method with LEHD is close to, but does not fully reach, the SL results for CVRP. We note a general gap of about 1% between the LEHD SL results for CVRP in the original paper [24] and our reproduced results, which we attribute to the slightly different training method we aligned with BQ. At the bottom, we group the non-greedy results of our trained BQ model using beam search and our sequence decoding method as an inference technique (coupled with Top-p sampling).

5.2 Job Shop Scheduling Problem

Problem setup The standard JSSP of size $J \times M$ is a CO problem with J jobs, each consisting of M operations with given processing times. Each job operation must run on exactly one of M machines (precedence constraint), which are assigned to the operations bijectively. A machine can only process one operation at a time. The op-

Table 1. Results for TSP and CVRP. 'bs' means beam search with a given width. The last two rows result from applying our proposed decoding method with Top-p sampling ($p = 0.95$ for TSP and 0.8 for CVRP). LEHD RL+SIL and GD SIL (LEHD) only report results on the TSP. Gaps are obtained with respect to Concorde [1] for TSP and HGS [37] for CVRP. Reported times are the duration of solving all instances.

Method	Test (10k inst.)		Generalization (128 inst.)				Test (10k inst.)		Generalization (128 inst.)			
	TSP $N = 100$		TSP $N = 200$		TSP $N = 500$		CVRP $N = 100$		CVRP $N = 200$		CVRP $N = 500$	
	Gap	Time	Gap	Time	Gap	Time	Gap	Time	Gap	Time	Gap	Time
AM, bs1024 [17]	2.49%	5m	6.18%	15s	17.98%	2m	4.20%	10m	8.18%	24s	18.01%	3m
POMO, augx8 [22]	0.14%	15s	1.57%	2s	20.18%	16s	0.69%	25s	4.87%	3s	19.90%	24s
SGBS [5]	**0.06%**	4m	**0.67%**	14s	11.42%	5m	**0.08%**	7m	**2.58%**	20s	15.34%	6m
MDAM, bs50 [41]	0.40%	20m	2.04%	3m	**9.88%**	11m	2.21%	25m	4.30%	3m	**10.50%**	12m
BQ SL, greedy [8]	0.40%	30s	**0.60%**	3s	**0.98%**	16s	3.03%	50s	**2.63%**	4s	**3.75%**	22s
GD SIL (BQ), greedy [30]	0.41%	30s	0.64%	3s	1.12%	16s	3.26%	50s	3.05%	4s	3.89%	22s
Ours (BQ), greedy	**0.37%**	30s	**0.60%**	3s	1.10%	16s	**2.96%**	50s	3.27%	4s	5.77%	22s
LEHD SL, greedy [24]	0.58%	25s	0.95%	2s	1.72%	11s	**4.26%**	40s	**3.77%**	6s	**4.36%**	12s
LEHD RL+SIL, greedy [24]	1.07%	25s	1.45%	2s	2.58%	11s	-	-	-	-	-	-
GD SIL (LEHD), greedy [30]	0.40%	25s	0.72%	2s	1.43%	11s	-	-	-	-	-	-
Ours (LEHD), greedy	**0.39%**	25s	**0.66%**	2s	**1.40%**	11s	5.08%	40s	4.60%	6s	5.51%	12s
Ours (BQ), bs64	0.004%	7m	0.04%	45s	0.33%	3m	1.07%	10m	1.48%	50s	3.40%	3m
Ours (BQ), $k = 64, s = 25$	0.003%	10m	0.04%	70s	0.22%	12m	0.42%	25m	0.62%	2m	2.69%	20m
Ours (BQ), $k = 128, s = 10$	**0.0009%**	35m	**0.02%**	4m	**0.18%**	50m	**0.14%**	50m	**0.27%**	6m	**2.19%**	50m

erations of a job must run in order. The objective is to find a feasible schedule that processes all operations of all jobs and has a minimum makespan. We follow the constructive formulation [6, 30] where an unfinished job is chosen of which to schedule the next ready operation at each iterative step. In particular, a feasible solution can be represented by a (not necessarily unique) sequence of jobs.

Data Random instances are generated in the standard way by uniformly sampling integer processing times from $[1, 99]$ and randomly permuting the M machines to determine the machine order for a job's operations. We generate a random validation set of 100 instances of size 20×20. We perform testing on the widely used benchmark dataset by Taillard [35].

Policy network architecture We use the recent architecture by Pirnay and Grimm [30], where the operations attend to each other through stacked transformer blocks, switching between different masking schemes. We refer to the appendix of [30] for a detailed architecture description. We use the same setup with six transformer blocks with eight heads and a hidden dimension of 256 in the feed-forward network. We note that the downside of the transformer-based architecture is its quadratic complexity with respect to the total number of operations.

Training We follow the training scheme of [30]. We train the model with SIL for 450 epochs. In each epoch, we decode solutions to 512 instances of size randomly chosen from $\{15 \times 10, 15 \times 15, 15 \times 20\}$. We use a beam width of $k = 64$ and a step size of $s = 50$. For the largest size 15×20, this takes about 5 minutes. As for the routing problems, we set the initial learning rate of the Adam optimizer to 2e-4, clipping gradients to unit norm. In each epoch, we train the model on 1,000 batches consisting of 512 subschedules each.

Baselines We compare our method to **(a)** L2D [42] and **(b)** ScheduleNet [29], two constructive RL approaches using graph neural networks, **(c)** L2S [43], a recent impressive deep RL guided improvement heuristic with 500 and 5000 improvement steps, **(d)** SPN [6], a SIL approach which samples sequences with replacement during training, and **(e)** GD [30], a SIL approach which shares the same network architecture with our model.

Results We summarize the results in Table 2. Our greedy results outperform the baseline greedy results and L2S with 500 steps (comparable runtime) by a wide margin. Notably, we achieve an improvement of $> 4\%$ on 30×20 compared to GD, which uses the same network architecture and already outperforms the other methods. We can further shrink the optimality gap by using our decoding approach as an inference technique (with Top-p sampling with $p = 0.9$). Because a solution to a JSSP instance of size $J \times M$ is a sequence of length $J \cdot M$, the results showcase the strength of our sequence decoding method, especially in tasks with a longer planning horizon.

5.3 Sampling comparison

Our decoding method relies on SBS to sample sequences WOR, which is already an established method to diversify the model output and enhance exploration. SBS is also the basis for the sampling method in GD [30]. Therefore, we compare the quality of the best solution obtained when decoding the policy with our method and sampling sequences WOR (with SBS) and GD. For each considered problem class TSP, CVRP, and JSSP, we take a checkpoint from the middle of the training process when the policy still has room for improvement and exploration is advantageous. We then decode solutions with beam width k and step size s using our method, and also with SBS and GD using the same computational budget. To ensure that we allow SBS and GD at least the same computational budget, we count the number of times we transition from a node in the search tree to a child node. One can show that when l is the length of a complete solution, our decoding method with parameters k and s takes $g(k, s)$ node transitions with

$$g(k, s) = k \cdot \left(tl - \frac{st^2 - st}{2} \right), \text{ where } t = \left\lceil \frac{l}{s} \right\rceil . \tag{3}$$

Sampling k sequences WOR with SBS takes $k \cdot l$ node transitions. In particular, we allow $k \cdot \lceil h(s) \rceil$ transitions for SBS and GD, where

$$h(s) = \frac{g(k, s)}{kl} = \frac{2tl - st^2 + st}{2l}. \tag{4}$$

For example, for $l = 100$, $k = 64$ and $s = 10$, we have $h(s) = 5.5$, so we grant SBS to sample $6k = 384$ sequences from the root. The

Table 2. JSSP results on the Taillard [35] dataset. Optimal gaps are computed with respect to the best solutions found in the literature by [29, 43, 6, 30]. Results in the bottom row are obtained by applying our proposed decoding method with Top-p sampling, with $p = 0.9$

.

	15 × 15		20 × 15		20 × 20		30 × 15		30 × 20		50 × 15		50 × 20		100 × 20	
Method	Gap	Time	Gap	Time	Gap	Time	Gap	Time	Gap	Time	Gap	Time	Gap	Time	Gap	Time
L2D, greedy [42]	26.0%	0s	30.0%	0s	31.6%	1s	33.0%	1s	33.6%	2s	22.4%	2s	26.5%	4s	13.6%	25s
ScheduleNet, greedy [29]	15.3%	3s	19.4%	6s	17.2%	11s	19.1%	15s	23.7%	25s	13.9%	50s	13.5%	1.6m	6.7%	7m
L2S, 500 steps [43]	9.3%	9s	11.6%	10s	12.4%	11s	14.7%	12s	17.5%	14s	11.0%	16s	13.0%	23s	7.9%	50s
SPN SIL, greedy [6]	13.8%	0s	15.0%	0s	15.2%	0s	17.1%	0s	18.5%	1s	10.1%	1s	11.6%	1s	5.9%	2s
GD SIL, greedy [30]	9.6%	1s	9.9%	1s	11.1%	1s	9.5%	1s	13.8%	2s	2.7%	2s	6.7%	3s	1.7%	28s
Ours, greedy	**7.7%**	1s	**8.5%**	1s	**8.7%**	1s	**8.4%**	1s	**9.6%**	2s	**2.2%**	2s	**4.9%**	3s	**1.0%**	28s
L2S, 5000 steps [43]	6.2%	1.5m	8.3%	1.7m	9.0%	2m	9.0%	2m	12.6%	2.4m	4.6%	2.8m	6.5%	3.8m	3.0%	8.4m
Ours, $k = 64, s = 50$	**3.0%**	10s	**4.1%**	25s	**4.1%**	1m	**3.9%**	90s	**6.2%**	5m	**0.4%**	10m	**1.7%**	30m	**0.1%**	5h

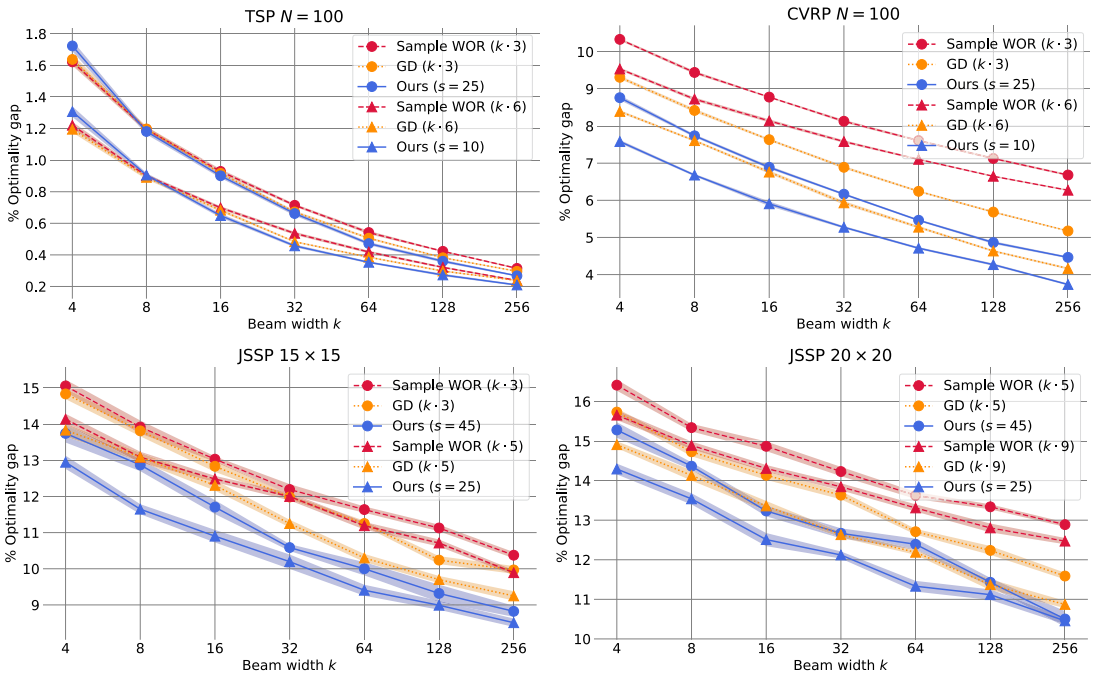

Figure 3. Decoding the policy with our sequence decoding method ('Ours') compared to sampling sequences without replacement with SBS ('Sample WOR') and to the sampling method GD [30]. The number of sequences sampled WOR and with GD are given as multiples of the beam width k to ensure alignment of the computational effort. Points with same marker mean same compute budget. For the routing problems, we average optimality gaps across 100 instances. For the JSSP, the corresponding Taillard benchmark set is used. Sampling for each data point is repeated ten times; shades denote standard errors.

same applies to GD, where we sample for $\lceil h(s) \rceil$ rounds with beam width k, using the constants for scaling the advantages in between rounds as reported in [30] (without nucleus sampling).

We show the decoding results in Figure 3. We observe only a small improvement over sampling WOR and GD for the TSP model, as the policy is already confident (< 2% optimality gap for $k = 4$). For the CVRP and the JSSP model, we see a significant improvement of about 1-2% over sampling WOR, showing that our method can consistently take advantage of the search budget.

6 Conclusion

The SIL paradigm, where the neural policy iteratively learns from its own decoded predictions, offers a promising path for NCO to overcome the training complexities and generalization challenges associated with RL methods. However, it requires the construction of a multitude of ever-improving solutions for a substantial number of problem instances during training. Despite this need, there needs to be more guidance apart from standard sequence decoding techniques from natural language processing on *how* to effectively build these solutions principled and exploratively. In this paper, we have proposed a novel sequence decoding technique for constructive NCO that is strikingly simple, does not rely on problem specifics, and works particularly well for longer planning horizons. We have achieved this by following a sampled, seemingly good solution for a limited number of steps and, importantly, replanning it by considering previously unseen alternatives. We have demonstrated our method on three prominent CO problems, showing comparable performance to training directly on expert solutions and the ability to surpass existing SIL methods. Notably, we have achieved new state-of-the-art results for NCO on the prominent JSSP Taillard benchmark. Due to its flexibility, our method can in principle also be used in other problem-specific SIL approaches, such as [25].

Acknowledgements

This work was funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) - 466387255 - within the Priority Programme "SPP 2331: Machine Learning in Chemical Engineering". The authors gratefully acknowledge the Competence Center for Digital Agriculture (KoDA) at the University of Applied Sciences Weihenstephan-Triesdorf for providing additional computational resources.

References

[1] D. Applegate, R. Bixby, V. Chvatal, and W. Cook. *The traveling salesman problem: a computational study*. Princeton University Press, 2006.

[2] T. Bachlechner, B. P. Majumder, H. Mao, G. Cottrell, and J. McAuley. Rezero is all you need: Fast convergence at large depth. In *Uncertainty in Artificial Intelligence*, pages 1352–1361. PMLR, 2021.

[3] I. Bello, H. Pham, Q. V. Le, M. Norouzi, and S. Bengio. Neural combinatorial optimization with reinforcement learning. *arXiv preprint arXiv:1611.09940*, 2016.

[4] Y. Bengio, A. Lodi, and A. Prouvost. Machine learning for combinatorial optimization: a methodological tour d'horizon. *European Journal of Operational Research*, 290(2):405–421, 2021.

[5] J. Choo, Y.-D. Kwon, J. Kim, J. Jae, A. Hottung, K. Tierney, and Y. Gwon. Simulation-guided beam search for neural combinatorial optimization. *Advances in Neural Information Processing Systems*, 35: 8760–8772, 2022.

[6] A. Corsini, A. Porrello, S. Calderara, and M. Dell'Amico. Self-labeling the job shop scheduling problem. *arXiv preprint arXiv:2401.11849*, 2024.

[7] M. Deudon, P. Cournut, A. Lacoste, Y. Adulyasak, and L.-M. Rousseau. Learning heuristics for the tsp by policy gradient. In *Integration of Constraint Programming, Artificial Intelligence, and Operations Research: 15th International Conference, CPAIOR 2018, Delft, The Netherlands, June 26–29, 2018, Proceedings 15*, pages 170–181. Springer, 2018.

[8] D. Drakulic, S. Michel, F. Mai, A. Sors, and J.-M. Andreoli. Bq-nco: Bisimulation quotienting for efficient neural combinatorial optimization. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023.

[9] Z.-H. Fu, K.-B. Qiu, and H. Zha. Generalize a small pre-trained model to arbitrarily large tsp instances. In *Proceedings of the AAAI conference on artificial intelligence*, volume 35, pages 7474–7482, 2021.

[10] J. Hare. Dealing with sparse rewards in reinforcement learning. *arXiv preprint arXiv:1910.09281*, 2019.

[11] A. Holtzman, J. Buys, L. Du, M. Forbes, and Y. Choi. The curious case of neural text degeneration. In *International Conference on Learning Representations*, 2020.

[12] A. Hottung and K. Tierney. Neural large neighborhood search for the capacitated vehicle routing problem. In *24th European Conference on Artificial Intelligence (ECAI 2020)*, 2020.

[13] A. Hottung, Y.-D. Kwon, and K. Tierney. Efficient active search for combinatorial optimization problems. In *International Conference on Learning Representations*, 2022.

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

[15] M. Kim, J. Park, et al. Learning collaborative policies to solve np-hard routing problems. *Advances in Neural Information Processing Systems*, 34:10418–10430, 2021.

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

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

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

[19] W. Kool, H. Van Hoof, and M. Welling. Stochastic beams and where to find them: The gumbel-top-k trick for sampling sequences without replacement. In *International Conference on Machine Learning*, pages 3499–3508. PMLR, 2019.

[20] W. Kool, H. Van Hoof, and M. Welling. Ancestral gumbel-top-k sampling for sampling without replacement. *Journal of Machine Learning Research*, 21(47):1–36, 2020.

[21] W. Kool, H. van Hoof, J. Gromicho, and M. Welling. Deep policy dynamic programming for vehicle routing problems. In *International conference on integration of constraint programming, artificial intelligence, and operations research*, pages 190–213. Springer, 2022.

[22] Y.-D. Kwon, J. Choo, B. Kim, I. Yoon, Y. Gwon, and S. Min. Pomo: Policy optimization with multiple optima for reinforcement learning. *Advances in Neural Information Processing Systems*, 33:21188–21198, 2020.

[23] J. Lee, S. Kee, M. Janakiram, and G. Runger. Attention-based reinforcement learning for combinatorial optimization: Application to job shop scheduling problem. *arXiv preprint arXiv:2401.16580*, 2024.

[24] F. Luo, X. Lin, F. Liu, Q. Zhang, and Z. Wang. Neural combinatorial optimization with heavy decoder: Toward large scale generalization. In *Thirty-seventh Conference on Neural Information Processing Systems*, 2023.

[25] F. Luo, X. Lin, Z. Wang, T. Xialiang, M. Yuan, and Q. Zhang. Self-improved learning for scalable neural combinatorial optimization. *arXiv preprint arXiv:2403.19561*, 2024.

[26] C. Meister, A. Amini, T. Vieira, and R. Cotterell. Conditional poisson stochastic beam search. *arXiv preprint arXiv:2109.11034*, 2021.

[27] M. Nazari, A. Oroojlooy, L. Snyder, and M. Takác. Reinforcement learning for solving the vehicle routing problem. *Advances in neural information processing systems*, 31, 2018.

[28] J. Park, J. Chun, S. H. Kim, Y. Kim, and J. Park. Learning to schedule job-shop problems: representation and policy learning using graph neural network and reinforcement learning. *International Journal of Production Research*, 59(11):3360–3377, 2021.

[29] J. Park, S. Bakhtiyarov, and J. Park. Schedulenet: Learn to solve multi-agent scheduling problems with reinforcement learning, 2022.

[30] J. Pirnay and D. G. Grimm. Self-improvement for neural combinatorial optimization: Sample without replacement, but improvement. *Transactions on Machine Learning Research*, 2024. ISSN 2835-8856. URL https://openreview.net/forum?id=agT8ojoH0X. Featured Certification.

[31] S. Rennie, E. Marcheret, Y. Mroueh, J. Ross, and V. Goel. Self-critical sequence training for image captioning. *IEEE CVPR*, 2017.

[32] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[33] K. Shi, D. Bieber, and C. Sutton. Incremental sampling without replacement for sequence models. In *International Conference on Machine Learning*, pages 8785–8795. PMLR, 2020.

[34] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, Y. Chen, T. Lillicrap, F. Hui, L. Sifre, G. van den Driessche, T. Graepel, and D. Hassabis. Mastering the game of go without human knowledge. *Nature*, 550: 354–359, 2017.

[35] E. Taillard. Benchmarks for basic scheduling problems. *European Journal of Operational Research*, 64:278–285, 1993.

[36] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[37] T. Vidal. Hybrid genetic search for the cvrp: Open-source implementation and swap* neighborhood. *Computers & Operations Research*, 140: 105643, 2022.

[38] O. Vinyals, M. Fortunato, and N. Jaitly. Pointer networks. *Advances in neural information processing systems*, 28, 2015.

[39] R. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8:229–256, 1992.

[40] Y. Wu, W. Song, Z. Cao, J. Zhang, and A. Lim. Learning improvement heuristics for solving routing problems. *IEEE transactions on neural networks and learning systems*, 33(9):5057–5069, 2021.

[41] L. Xin, W. Song, Z. Cao, and J. Zhang. Multi-decoder attention model with embedding glimpse for solving vehicle routing problems. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pages 12042–12049, 2021.

[42] C. Zhang, W. Song, Z. Cao, J. Zhang, P. S. Tan, and X. Chi. Learning to dispatch for job shop scheduling via deep reinforcement learning. *Advances in Neural Information Processing Systems*, 33:1621–1632, 2020.

[43] C. Zhang, Z. Cao, W. Song, Y. Wu, and J. Zhang. Deep reinforcement learning guided improvement heuristic for job shop scheduling. In *The Twelfth International Conference on Learning Representations*, 2024.

[44] L. Zhao, W. Shen, C. Zhang, and K. Peng. An end-to-end deep reinforcement learning approach for job shop scheduling. In *2022 IEEE 25th International Conference on Computer Supported Cooperative Work in Design (CSCWD)*, pages 841–846, 2022.