Cost Shaping via Reinforcement Learning for Vehicle Routing Problems

Yeong-Dae Kwon[1] **Jinho Choo**[1] **Daseul Bae**[1] **Jihoon Kim**[1] **André Hottung**[2]

[1]Samsung SDS, Korea [2]Bielefeld University, Germany

Abstract

This document describes TEAM_SB's submission to the EURO Meets NeurIPS 2022 Vehicle Routing Competition. [1]

1 Static Problems

For static problems, our initial approach was to train a deep neural net model that constructs a valid solution in an autoregressive way [2, 3]. We started with VRPTW problem instances with 100 nodes and succeeded in building a model that produces a solution in one shot (instantly), with its average solution quality having roughly 5% gap from that of the baseline Hybrid Genetic Search (HGS) algorithm provided by the competition organizers. While there exist additional improvement schemes [4, 5] that can utilize the given time limit to search for higher quality solutions, we decided that 5% gap was yet too large to overcome. We then switched our focus to modifying the HGS algorithm instead.

Here we list the four major modifications to HGS we have made for our static-problem solver. The first two contribute to the improvement of the solution qualities significantly. The last two have shown marginal (but positive) effects in our tests, but they are included because we find these ideas interesting and worth further refining in the future.

1.1 Parameter Tuning for Different Problem Sizes

The HGS code contains many configuration parameters whose values are meant to be adjusted empirically for the best performance. Their optimal values should depend on the size of the problem (*i.e.*, `nbClients`, the number of clients), especially since the evaluation protocol for the competition applies different time limits according to the values of `nbClients`. Hence, we have manually tested many different values for a few critical parameters, and have settled on the following values:

- `config.minimumPopulationSize` : 40 / 50 / 50
- `config.generationSize` : 45 / 40 /40
- `config.nbIter` : 20000 / 30000 / 40000

where the notation 'A / B / C' means that A is for instances with `nbClients` $<$ 300, B is for 300 $\leq$ `nbClients` $<$ 500, and C is for 500 $\leq$ `nbClients`. Note that the increased population sizes due to these changes go hand in hand with our altered parents selection strategy explained below.

1.2 Parents Selection

The 'binary tournament selection' scheme, used in the original baseline code, can be considered as a systematic way to assign a biased probability to each candidate in the population pool for being

EURO Meets NeurIPS 2022 Vehicle Routing Competition Workshop.

selected as a parent (to generate offspring). The underlying rule is that the better the fitness value, the bigger the chance a candidate has for it to be chosen as a parent. Through the binary tournament process, the selection probability distribution becomes a linear function of the rankings of the fitness values. We want to optimize the shape of this distribution.

A new, explicit, analytic formula would have been nicer, but we choose to make the minimal change to the original code and keep the tournament-style selection scheme, instead. We replace the binary tournament with the k-way tournament selection scheme, where k can be larger (smaller) than 2, which makes the probability distribution steeper (flatter). More specifically, for Parent-1, k is 4, and it is changed to 2 when 'nbIterNonProd' reaches 50% of 'nbIter'. For Parent-2, k begins with 4, and it is changed to 3, 2, and 1 as 'nbIterNonProd' reaches 25%, 50%, and 75% of 'nbIter' respectively. Here, 'nbIter' is the allowed number of search iterations with no improvement on the incumbent solution (before a reset) and 'nbIterNonProd' is the current count on such iterations. (Both terms are the variable names used in the original code provided.) Basically, we initially give higher chances to 'good' candidates to be selected as a parent compared to the original code, but when they fail to make a real progress repeatedly, the 'bad' candidates are given increasingly more chances.

While our method described here does not involve ML yet, replacing this manually-tuned selection strategy with a RL-trained policy (for choosing the optimal k values, for example) would be desirable, similarly to how we have constructed the dynamic problem solver using neural networks (to be explained in the next section).

1.3 Telomeres

In many Genetic Algorithm (GA) variants, the two parents are eliminated from the genetic pool after they produce offspring. HGS does not have this feature, and the parents remain in the population pool as long as they have strong enough fitness. Limiting the lifetime of a 'gene' can increase the diversity of the pool. We keep a record of children counts for each member of the population so that when one of them has produced more than a certain maximum number (=100) of offspring, it is removed from the pool, regardless of its fitness.

1.4 Restart from the Past

HGS restarts the search process when it cannot produce a better solution over a certain number of iterations. And when it restarts, it begins with new, randomly generated population, which takes a while to turn into the one with decent solutions. Hence, when there is little time left, restarting with a fresh random state is not so sensible. To mitigate this issue, we save copies of the population pool regularly. When HGS needs to restart but has spent more than half of the allowed runtime, we begin with one of these saved copies, depending on the amount of time left.

2 Dynamic Problems

Our basic approach for dynamic problems is of a LAZY-type, in that we prefer to defer whenever possible. We use the following **two-step algorithm** in deciding what routes to return to the environment at each epoch of the problem:

1. We first generate routes using all the clients given for the current epoch (as though we are solving a static problem) using the modified HGS algorithm explained below.
2. Among all the generated routes, we return only those that contain one or more clients marked as 'must-dispatch,' and the routes having no 'must-dispatch' nodes are ignored.

Neural networks trained by reinforcement learning (RL) are used in both steps.

2.1 Modified HGS Cost Function, v1.0

Assume that for a given epoch, there are N clients, where some of them have the 'must-dispatch' statuses and the others do not. We run HGS with all N clients. During its search process, let's say that HGS creates a number of routes (τ_1, τ_2, ..., τ_a) all of which contain at least one 'must-dispatch' client, and b number of routes ($\bar{\tau}_1$, $\bar{\tau}_2$, ..., $\bar{\tau}_b$) that contain no 'must-dispatch' client. We want to treat

those two groups differently. The original cost function of HGS (*i.e.*, the total driving duration of all routes) can be re-written as

$$\text{Cost}_{\text{original}} = \sum_{i=1}^{a} d_i + \sum_{j=1}^{b} \bar{d}_j \tag{1}$$

where d_i is the driving duration of τ_i and $\bar{d}_j$ is the driving duration of $\bar{\tau}_j$.

As our strategy is to defer all the clients contained in $\bar{\tau}_1$, $\bar{\tau}_2$, ..., $\bar{\tau}_b$, we are not interested in minimizing $\bar{d}_j$'s. Hence, we modify the internal cost function of HGS into

$$\text{Mod-Cost}_{\text{v1.0}} = \sum_{i=1}^{a} d_i / \sum_{i=1}^{a} n_i \tag{2}$$

where n_i is the number of clients contained in τ_i. The division by the sum of n_i's makes the cost defined as **the averaged driving duration per client**, rather than just the total driving duration for that epoch. Without this denominator, HGS will simply ignore all the none-'must-dispatch' clients to minimize $\sum_{i=1}^{a} d_i$, which is equivalent to the baseline LAZY strategy provided by the organizers.

Note that when $b = 0$, as in the last epochs of the dynamic problems,

$$\text{Cost}_{\text{original}} = N \cdot \text{Mod-Cost}_{\text{v1.0}} \tag{3}$$

so that the minimization of the both cost functions are equivalent to each other.

2.2 Modified HGS Cost Function, v2.0 with 'Priorities'

We further modify the HGS cost function by changing its denominator. Since the denominator of Mod-Cost$_{\text{v1.0}}$ (Eq. (2)) is just a head count, every clients contained in τ_1, τ_2, ..., τ_a contribute to the size of the denominator equally. We change this, by assigning a 'priority value' p_e to each client e and switching the cost function to

$$\text{Mod-Cost}_{\text{v2.0}} = \sum_{i=1}^{a} d_i / \sum_{i=1}^{a} \left(\sum_{e \in \tau_i} p_e \right). \tag{4}$$

Note that when all p_e's are 1, this is equivalent to Equation (2). Also, when $b = 0$, minimizing Mod-Cost$_{\text{v2.0}}$ is still equivalent to minimizing Cost$_{\text{original}}$, regardless of the values of p_e's.

When HGS tries to minimize this new cost function, the clients with high priority values are more likely to be included in the τ_i's and be dispatched rather than be deferred at the current epoch. Hence, by adjusting the assignment of the priority values, one can now externally manipulate the output of HGS. For example, one may handcraft a priority function in a way that low priority values are assigned to the clients with larger 'window-closing-time' parameters. This strategy generally leads to better results, as these clients are now more easily deferred at earlier epochs, and more likely to be dispatched at later, better-suited epochs. However, exactly how much low (or high) priority values to use in order to achieve the optimal solver performance is difficult to obtain analytically.

Rather than handcrafting (and manually tuning) a high-performing priority function, we use an ML approach to create one automatically. We train a deep neural net so that it can tell us which priority value to use for each client in order to achieve the best final result (the shortest total driving time) at the end of all epochs. The details of the neural network will be explained in the next subsection, but our neural net approach is shown to **decrease the total driving time by about 10%** compared to having all p_e's set to 1.

2.3 Neural-Net-Based Cost Shaping

Our neural network has the architecture of 5-layer perceptron (MLP) with ReLU activations. It operates on each individual client and sets its priority value. Input vectors to the neural net contain information of the client, such as how much time (and how many epochs) are left before that client becomes 'must-dispatch.' The outputs of the neural net are the selection probabilities for the elements of a set of priority values, {0.2, 0.4, 0.6, 0.8, 1.0, 1.5, 2.0}. Priorities for 'must-dispatch' clients are fixed to 1.0.

To train our model, we have built a customized environment, similar to the one provided by the competition organizers, but capable of running multiple dynamic VRPTW problem instances in parallel. For each problem instance, we solve it in 10 different ways (in parallel) based on the sampled outputs from our neural model. Then the average of the resulting 10 total traveling distances is used as a baseline for REINFORCE training in adjusting the neural parameters [6, 3]. Our training batch size is 64, meaning that the parameters are repeatedly updated after solving 64×10 VRPTW training instances. Only those public 250 VRPTW instances provided by the organizers have been used for training. (200 of them, to be exact, as we have used the other remaining 50 as a validation set to monitor the progress of our training.)

The most time-consuming part of our training process is, of course, running HGS for each epoch of each dynamic problem instance. To save time, we limit the runtime of HGS to [5, 30] seconds, depending on the number of clients it handles at each epoch. The model we use in our submission code has been trained for 20 hours on our server with 4 GPUs and 60+ CPUs.

2.4 Designation of Extra 'Must-Dispatch' Nodes by Second Neural Network

In addition to the neural network that selects priorities, we employ a second neural network whose job is to confer the 'must-dispatch' status to the clients who are not really the genuine 'must-dispatch' type (meaning that deferring them would not lead to infeasible solutions). Conceptually, this is similar to setting the priority values of those clients to a very big number, which forces them to be dispatched immediately. But such assignment of exceptionally large priority values to a few select clients from time to time brings imbalance to the the distribution of $\sum_{e \in \tau_i} p_e$ and makes the RL training difficult. So we have separately trained a second, independent neural network to output probabilities for {`True, False`} that determines whether to set the status of a client to be 'must-dispatch' or not.

This second neural network is also trained by REINFORCE, in the same way that our first neural network is trained. The goal of the training is to locate the additional clients to be marked as 'must-dispatch' that eventually lead to higher quality solutions when our two-step dynamic VRPTW algorithm is executed. In our tests, this second neural net further decreases the total traveling distances of the solutions by 0.5%, on average.

2.5 Further Improvement Ideas (that we regret not having had enough time for)

Most importantly, the performance of our neural networks is limited by its narrow scope. At the moment, their decisions are made locally, not knowing how the priorities are set on other clients. If the information of all clients as a whole is used as an input, rather than the individual ones, we believe that a neural network (equipped with Transformer-like structures similar to MatNet [7]) can produce better solutions, reflecting the global information of the given instances.

Having to have a pre-defined, discrete set of candidates for the priority values is also something to further improve, as we aim to minimize manual settings of hyper-parameters as much as possible and make the construction process of our solvers near-automatic. It will also improve the solver's performance if the neural net can be made to return a priority value directly (rather than probabilities) chosen from a continuous set of real numbers.

3 Code

Our code submitted for the competition is publicly available at `https://github.com/yd-kwon/NeurIPS-2022-VRPTW-Competition`.

References

[1] Wouter Kool, Laurens Bliek, Danilo Numeroso, Robbert Reijnen, Reza Refaei Afshar, Yingqian Zhang, Tom Catshoek, Kevin Tierney, Eduardo Uchoa, Thibaut Vidal, and Joaquim Gromicho. The EURO meets NeurIPS 2022 vehicle routing competition. 2022.

[2] Wouter Kool, Herke van Hoof, and Max Welling. Attention, learn to solve routing problems! *International Conference on Learning Representations*, 2019.

[3] Yeong-Dae Kwon, Jinho Choo, Byoungjip Kim, Iljoo Yoon, Youngjune Gwon, and Seungjai Min. Pomo: Policy optimization with multiple optima for reinforcement learning. *Advances in Neural Information Processing Systems 33*, 2020.

[4] André Hottung, Yeong-Dae Kwon, and Kevin Tierney. Efficient active search for combinatorial optimization problems. *International Conference on Learning Representations*, 2022.

[5] Jinho Choo, Yeong-Dae Kwon, Jihoon Kim, Jeongwoo Jae, André Hottung, Kevin Tierney, and Youngjune Gwon. Simulation-guided beam search for neural combinatorial optimization. *Advances in Neural Information Processing Systems 35*, 2022.

[6] Wouter Kool, Herke van Hoof, and Max Welling. Buy 4 reinforce samples, get a baseline for free! In *DeepRLStructPred@ICLR*, 2019.

[7] Yeong-Dae Kwon, Jinho Choo, Iljoo Yoon, Minah Park, Duwon Park, and Youngjune Gwon. Matrix encoding networks for neural combinatorial optimization. *Advances in Neural Information Processing Systems 34*, 2021.