A modified HGS for static VRPTW and a neighbors addition-postponement strategy for dynamic VRPTW

Juan Pablo Mesa*
Universidad EAFIT
Medellín, Colombia
jmesalo@eafit.edu.co

Alejandro Montoya
Universidad EAFIT, Coordinadora Mercantil
Medellín, Colombia
jmonto36@eafit.edu.co

Alejandro Uribe
Universidad EAFIT
Medellín, Colombia
auribev1@eafit.edu.co

Camilo Álvarez
Universidad EAFIT
Medellín, Colombia
acalvarezv@eafit.edu.co

Abstract

This document describes HowToRoute's submission to the EURO Meets NeurIPS 2022 Vehicle Routing Competition. [1] For the static version, we applied two modifications: to the local search and to the offspring generation procedure. For the dynamic version, our method is based on two procedures: adding non-obligatory customers and postponing non-obligatory customers that worsen the solution. Our methods achieved an average improvement of 0.5% for the provided instances and 0.15% for the leaderboard instances in the static variant. Similarly, they delivered solutions that cost, on average, 10.7%, 19.6%, and 37.1% less than the greedy, random, and lazy baselines, respectively, in the dynamic variant. The proposed methods ranked 3rd in the qualification phase of the competition.

1 Introduction

This document describes the proposed methods to solve the Vehicle Routing Problem with Time Windows (VRPTW) for the static and dynamic versions of the EURO Meets NeurIPS 2022 Vehicle Routing Competition [1]. This paper is structured as follows: first, we explain the proposed method for the solution of the static problem. Then, the proposed solution for the dynamic problem is explained. Finally, we will discuss the results and conclusions of our work.

2 Static method

The static method of our implementation is based on the Hybrid Genetic Search extension for VRPTW and dynamic requests [2] that was provided as a baseline. We performed several modifications to try to improve its performance. We applied two main modifications: to the local search and to the offspring generation procedure.

2.1 Local search modification

The SWAP* algorithm introduced in [3] and the Relocate*, with modifications for supporting time windows, are two of the neighborhoods from the local search. They involve operations between route pairs, and they are computationally expensive. Therefore, for each pair of considered routes,

*Corresponding author

EURO Meets NeurIPS 2022 Vehicle Routing Competition Workshop.

these neighborhoods are only applied with a certain intensification probability θ and when their circle sectors overlap.

The modification consists of decreasing the probability of application. Given an intensification probability θ, we multiply it by a percentage β, reducing the probability θ of exploring the SWAP* or Relocate* neighborhoods. This aims to reduce the number of evaluations of these computationally expensive neighborhoods during the local search, speeding up the convergence towards lower-cost solutions.

2.2 Offspring generation modification

The baseline genetic search implementation has two Crossover operators to generate Individuals from a pair of Parents, the order crossover (OX) and the Selective Route Exchange (SREX). Hvattum proposed an adjustment of the OX crossover that improves the performance of HGS on CVRP instances [4]. This new crossover operator is called Adjusted Order Crossover (AOX).

We implement the AOX without removing the original OX operator. Then our method now has three crossover operators: OX, AOX, and SREX. This leads to the problem of deciding which crossover operator to use for generating new individuals.

In the genetic search baseline, an individual is generated with the OX and SREX operators. For each crossover operator: First, two non-identical parents are selected from a pool of individuals, and the crossover operator is used to create a new offspring. Then, a new pair of non-identical parents is selected, and the process is repeated using the same crossover operator. Out of the four new offspring, the best one in terms of penalized cost is selected as the individual for the next phase. The final phase consists in applying local search to the new offspring.

In our method, we extend the number of offspring created for an individual generation with two more offspring. Therefore, for each new population individual that we want to generate, we create 6 offspring: two offspring with OX, two more with SREX, and the final two with AOX. The best offspring out of the 6 goes into the local search phase. The others are discarded.

3 Dynamic method

The method for solving the dynamic problem is based on two procedures: adding non-obligatory customers and postponing non-obligatory customers that worsen the solution. These procedures are performed sequentially for each epoch. In the rest of this section, we describe the steps of each procedure.

3.1 Adding non-obligatory customers

For each epoch of the dynamic problem, we consider the list of customers C, which is composed of must-dispatch or obligatory customers C_m and non-obligatory customers C_{no}. For each customer $c \in C$, we know its time window $[a_c, b_c]$. Algorithm 1 describes the main process for adding non-obligatory customers. The input parameters of this algorithm are described as follows: k is the number of near customers to consider (must dispatch or non-obligatory) of a given must dispatch customer, α is a time parameter to be used in the comparison of time windows (its use will be explained later) and i_{max} is the maximum number of iterations. In line 3, the algorithm starts by calling the NEIGHBORSADDING procedure which will be explained later. This procedure outputs a list of neighbors L to add for routing. These non-obligatory customers are added to the set of must-dispatch customers C_m in the list C_r, which are the customers to be routed (line 4). Then a cycle starts with the objective of adding customers but each time decreasing the number of neighbors to be added. This cycle ends when the set of customers to be routed C_r is equal to the total set of customers C or when the maximum iterations i_{max} have been reached. In line 6, the parameter k decreases the number of neighbors to consider. In line 7, we call again the NEIGHBORSADDING procedure with the set of C_r customers updated with the new neighbors. Finally, the algorithm returns the set of customers to be routed C_r (in line 11).

Algorithm 2 explains the NEIGHBORSADDING function, which is used in Algorithm 1. The algorithm starts in line 2 by initializing the empty list of customers L. Then we iterate for each customer c in the list of customers to be routed C_r. In line 4, we call the NEARESTNEIGHBORS procedure that obtains

Algorithm 1 Main algorithm for adding non-obligatory customers

```
1: function AddingCustomers(k, α, i_max, C)
2:     i ← 0
3:     L ← NeighborsAdding(k, α, C_m)
4:     C_r ← C_m ∪ L
5:     while |C_r| ≠ |C| or i < i_max do
6:         k ← k − 1
7:         L ← NeighborsAdding(k, α, C_r)
8:         C_r ← C_r ∪ L
9:         i ← i + 1
10:    end while
11:    return C_r
12: end function
```

the k nearest neighbors of the customer to be routed $c \in C_r$ in terms of the round-trip time. This procedure returns the set N of nearest customers. In line 6, we evaluate whether the upper window of neighbor b_n is less than or equal to the upper window of customer b_c plus an additional time α. If this condition is met, customer c is included in the customer list L (as shown in line 7). This seeks to ensure that the neighbors that are added have an upper time window limit close to the upper time window limit of the customer to be routed c. Finally, the algorithm returns the list L.

Algorithm 2 Adding neighbors that meet time window constraint

```
1: function NeighborsAdding(k, α, C_r)
2:     L ← ∅
3:     for each c ∈ C_r do
4:         N ← NearestNeighbors(c, k)
5:         for each n ∈ N do
6:             if b_n ≤ b_c + α then
7:                 L ← L ∪ {n}
8:             end if
9:         end for
10:    end for
11:    return L
12: end function
```

3.2 Postponement of customers that worsen the solution

With the static solver, we route the C_r customers obtained from the previous algorithm. This allows us to obtain the solution S corresponding to a set of routes $r \in S$. Algorithm 3 describes the procedure to postpone customers. The parameter entered to the function is ω, which is used in the comparison between the cost saved by postponing or removing a customer with the average cost of the routes. The procedure starts in line 2, where we iterate over the routes of the solution S. Then we call the AverageCost function that allows us to estimate the average cost of the route per customer a_{cost} (i.e., the total cost of the route divided by the number of customers in the route). In line 4, we iterate for each customer of the route $c \in r$. In each iteration, we call the SavedCost function to calculate the cost s_{cost} that the route would save in case of removing the customer c. In line 6, we evaluate whether s_{cost} is greater than or equal to the average cost of routing a_{cost} multiplied by ω and if customer c is not part of the initial set of must dispatch C_m. If these conditions are met, then customer c is removed from the set of customers to be routed C_r. Finally, the new list of customers C_r is sent to routing with the static solver.

4 Computational experiments and results

We evaluated the proposed methods in the real-world instances provided by the competition. We first describe the experiment's setup. Then, we discuss the parameter values used throughout the

Algorithm 3 Removing non-must dispatch customers that worsen the solution

```
function REMOVINGCUSTOMERS(S, ω, C_r)
    for each r ∈ S do
        a_cost ← AVERAGECOST(r)
        for each c ∈ r do
            s_cost ← SAVEDCOST(c)
            if s_cost ≥ a_cost × ω and c ∉ C_m then
                C_r ← C_r \ c
            end if
        end for
    end for
    return C_r
end function
```

Table 1: Static variant results.

	Baseline	Our Method
Provided instances	164,800	163,900
Leaderboard instances	180,840	180,565

experiments. Finally, we present and discuss the numerical results obtained using our method on the static and dynamic variants.

4.1 Static variant

For the instances provided by the competition. In the static variant, we run the solver for each instance for 240 seconds (4 minutes). In the dynamic variant, we set a time limit of 60 seconds (1 minute per epoch). For the leaderboard instances, the solver is executed with the time limit conditions of the competition.

In the case of the static version, we tuned the parameter β, and we found that the best value corresponds to $\beta = 0.2$. As previously mentioned, we generate 2 additional offspring each time an individual is created. With this configuration, we obtained the results shown in Table 1.

In the provided instances, with the baseline method, we obtained an average solution cost of $164,800 \pm 100$. Meanwhile, with our method, we obtained an average route cost of $163,900 \pm 100$. This is an average improvement of 0.5%. In the leaderboard instances, the baseline method obtained an average solution cost of $180,840 \pm 10$. While our method obtained an average cost of $180,565 \pm 10$. This represents an average improvement of 0.15%.

4.2 Dynamic variant

In the case of the dynamic version, we tuned the parameters with different values for each epoch. If it is the first epoch, we set $k = 4$. Otherwise $k = 8$. The parameter i_{max} varies according to the size of the set C_m. If C_m has more than 150 customers, then $i_{max} = 5$; if C_m has between 100 and 150 customers, then $i_{max} = 4$; otherwise, $i_{max} = 3$. The other parameters use the same values through all the epochs: $\alpha = 2$ and $\omega = 1.1$. With this configuration, we obtained the results shown in Table 2.

This configuration allowed us to find an average solution cost of $379,040 \pm 50$ for the provided instances and an average solution cost of $349,115 \pm 50$ for the leaderboard instances. We compared the performance, on the leaderboard instances, of our dynamic strategy with respect to the three initial

Table 2: Dynamic variant results.

	Lazy baseline	Random baseline	Greedy baseline	Our method
Provided instances	525,470	464,510	419,200	379,040
Leaderboard instances	555,331	434,528	391,197	349,115

baseline strategies provided by the competition: greedy, random, and lazy baselines. Our strategy delivered solutions that cost, on average, 10.7%, 19.6%, and 37.1% less than the greedy, random, and lazy baselines, respectively.

5 Conclusions

In this work, we have presented two methods to solve static and dynamic variants of the VRPTW. For the static variant, we applied two modifications to the baseline genetic search: a modification of the local search and a modification of the offspring generation procedure. For the dynamic variant, we proposed a method based on adding non-obligatory customers and postponing customers that worsen the solution.

The proposed methods were tested on the EURO Meets NeurIPS 2022 Vehicle Routing Competition instances. For the static variant, our method obtained an average improvement of 0.5% and 0.15% for the provided and leaderboard instances, respectively. For the dynamic variant, our method obtained improvements between 10 to 37% with respect to the provided baselines.

The results of our static method led us to 2nd place in the static performance rank of the leaderboard. Similarly, the results of our dynamic strategy led us to 6th place in the dynamic performance rank of the leaderboard. With these results, we obtained 3rd place on the overall rank of the qualification phase of the EURO Meets NeurIPS 2022 Vehicle Routing Competition.

In conclusion, our methods demonstrated good performance for both static and dynamic variants of the VRPTW problem. The improvements in the static solver are simple yet effective. Additionally, the strategies used in the dynamic variant are simple and can be easily extended into other variants of routing problems. Furthermore, each strategy can be integrated on its own into other more sophisticated methods to improve overall performance.

Acknowledgments and Disclosure of Funding

The authors acknowledge supercomputing resources made available by the Universidad EAFIT scientific computing center (APOLO) to conduct the research reported in this work and for its support for the computational experiments.

References

[1] W. Kool, L. Bliek, D. Numeroso, R. Reijnen, R. R. Afshar, Y. Zhang, T. Catshoek, K. Tierney, E. Uchoa, T. Vidal, and J. Gromicho, "The EURO meets NeurIPS 2022 vehicle routing competition," 2022.

[2] W. Kool, J. O. Juninck, E. Roos, K. Cornelissen, P. Agterberg, J. van Hoorn, and T. Visser, "Hybrid genetic search for the vehicle routing problem with time windows: a high-performance implementation," in *12th DIMACS Implementation Challenge Workshop*, 2022.

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

[4] L. M. Hvattum, "Adjusting the order crossover operator for capacitated vehicle routing problems," *Computers & Operations Research*, vol. 148, p. 105986, 2022.