Solving a static and dynamic VRP with time windows using hybrid genetic search and simulation

Jasper van Doorn
Department of Mathematics
Vrije Universiteit Amsterdam
Amsterdam, The Netherlands
j.m.h.van.doorn@vu.nl

Leon Lan
Department of Mathematics
Vrije Universiteit Amsterdam
Amsterdam, The Netherlands
l.lan@vu.nl

Luuk Pentinga
Department of Operations
University of Groningen
Groningen, The Netherlands
l.pentinga@rug.nl

Niels A. Wouda
Department of Operations
University of Groningen
Groningen, The Netherlands
n.a.wouda@rug.nl

Abstract

This short paper briefly describes OptiML's methods used to solve the two variants of the Euro meets NeurIPS 2022 vehicle routing competition [1]. Our methods ranked second in the qualification phase of the competition. The full software implementation of our ideas is available at `https://github.com/N-Wouda/Euro-NeurIPS-2022/`.

1 Introduction

The Euro meets NeurIPS vehicle routing competition introduces two problem variants that our solver must solve well. The first, *static* variant is a traditional VRP with hard time windows and no fleet limitation. The second, *dynamic* variant also considers a VRP with hard time windows, but now customer requests arrive in epochs. Every customer request has a time window, and can either be dispatched now or be postponed to a later epoch (if the time window allows). The dynamic solver makes these dispatch decisions. It then solves the resulting static VRPTW instance with the requests it wants to service in the current epoch.

In the following two sections we describe our approach for solving the static and dynamic variants of the competition problem.

2 Static variant

Our solver for the static VRPTW builds on the `hgs_vrptw` baseline [2, 3].

What we removed from the `hgs_vrptw` baseline Our solver simplifies the baseline implementation substantially. First, we removed the ordered exchange crossover and removed the 'giant tour' representation, storing only routes in our individuals. This also meant we could remove the linear-time split algorithm, resulting in a significant simplification at no cost in performance.

We also removed the constructive heuristics that were used to create an initial population. Instead, we create an initial population of random individuals. This ensures we quickly commence with the main genetic search loop, allowing our solver to also be used when there are very tight time limits. Additionally, it meant we could remove seven parameters related to these constructive heuristics.

Further, we removed the dynamic growth of the granular neighbourhood and minimum population sizes. Limited experimentation suggested growing these was ineffective, and removing them allowed us to remove five parameters that controlled the rate of growth.

We modified the local search part of the algorithm substantially (more on this below). As part of this modification, we removed the circle sector restrictions on the route operators. This meant we could remove another two parameters.

Finally, besides simplifying the baseline implementation, we also introduced Python bindings to the C++ codebase. These bindings allow us to directly interact with the solver from Python, which is much more efficient than writing and reading files.

Diversity and crossover From a population of individuals, two individuals are selected as parents and a crossover operator is applied to generate a new offspring solution. The diversity of the parents is key to a good offspring. A lack of diversity means the offspring will be too similar to the parents, and thus not provide us with a valuable new solution. On the other hand, a too diverse set of parents means the offspring often does not inherit the best qualities of either parent, and is nowhere near competitive and requires very long educate steps to improve. Based on these observations, we modified the parent selection mechanism slightly:

1. The first parent is selected by binary tournament.
2. We also select the second parent by binary tournament, but with a constraint on the diversity. In particular, we impose a constraint on the percentage of broken edge pairs between the two parents: this percentage needs to be between some lower and upper bounds $[\text{div}_l, \text{div}_u]$. Tuning suggests $\text{div}_l = 10\%$ and $\text{div}_u = 50\%$ are effective.

This simple tweak helped improve the performance of our solver substantially.

We removed the ordered exchange crossover method, since it did not offer any additional value given that we already have the SREX crossover method [4]. We tried many other crossover operators as well, including edge assembly crossover, broken pairs exchange, alternating exchange, and slack-induced sub-string exchange. None of these appeared to be significantly better than SREX.

Education and local search To produce a competitive new individual, the offspring generated during crossover is educated using local search. Our local search implementation applies regular, node-based local search during this education phase. We use the same operators as the baseline, but re-implement them using a general (N, M)-Exchange operator [5, 6]. This allowed us to re-use a lot of code, and implement a number of small (code) optimisations that sped up the operators significantly. We do not apply the intensifying route-based `RELOCATE*` or `SWAP*` operators during this phase.

We intensify only when the educated individual is a new global best solution. In that case we apply `RELOCATE*` and `SWAP*` to all route pairs, until there is no further improvement. We also apply sub-path enumeration: within each route, we fully enumerate k-node sub-paths to find improving permutations. Tuning suggests $k = 7$ is effective.

Parameter tuning Although our implementation has fewer parameters than the baseline method, tuning remained a significant challenge. We initially attempted to tune our solver using the SMAC3 algorithm [7], but did not find this to be effective in practice: the differences between various 'reasonable' parameter settings is so small that an automated algorithm has a difficult time effectively differentiating 'good' from 'better'.

We instead opted for a more prescriptive approach. First, we derive reasonable lower and upper bounds for the values of each parameter. Then, we split the parameters into three groups: (1) those related to penalty management, (2) those related to population management, and finally, (3) those related to local search. We generated 99 parameter configurations for the first group using a Latin hypercube design, and added the default setting as a control. We then solved each instance ten times using a different seed each, for each of the 100 configurations. We then took the best configuration and made that the default for the first group. With this default, we then repeated this approach for the second group, and finally, for the third group. Tuning the parameters in this fashion resulted in a respectable performance improvement.

3 Dynamic variant

Our solver for the dynamic VRPTW relies on simulating future requests.

Notations and definitions Let R_t denote the *epoch instance* at decision epoch t, i.e., the set of available requests at epoch t, consisting of newly arrived requests and requests that were postponed in the previous epoch. The epoch instance can be partitioned into a set of *must-dispatch* requests R_t^{must} and a set of *optional* requests R_t^{opt}. The optional requests do not have to be dispatched in the current epoch and may be postponed to future epochs. After having decided which of the optional requests to postpone, denoted by R_t^{post}, the remaining requests form the *dispatch instance* $R_t^{\text{disp}} = R_t \setminus R_t^{\text{post}}$. The dispatch instance is solved as a static VRPTW and the resulting solution S_t is submitted to the environment.

Using simulations to postpone optional requests We use simulations to determine which optional requests we should postpone. Given the epoch instance R_t, we sample a set of future requests $\widetilde{R}_t$. The sampling is determined by two new parameters: the number of *lookahead* epochs and the number of sampled requests per epoch. The epoch instance R_t together with the sampled requests $\widetilde{R}_t$ together form a *simulated instance*. The requests from R_t have no release dates, whereas the requests from $\widetilde{R}_t$ have release dates corresponding to the epoch they were sampled from. We solve the simulation instance very fast (less than 0.5 seconds) using our static solver. From the resulting solution, we count for every request if it is placed on a route with a must-dispatch request. If that is the case, we count it as a dispatch action, otherwise we count it as a postponement action.

We repeat this simulation procedure N^{sim} times. We then find all optional requests that were postponed in at least $H_t\%$ simulations, where H_t is an epoch-specific threshold value. We remove these requests from the epoch instance to obtain the dispatch instance, which we can then solve and submit to the environment.

Iterative application of the simulation method In each epoch, we had 60 seconds to submit a solution during qualification phase, and 120 seconds to submit a solution during the final phase. We obtained good results with the simulation method using only 15 seconds for the simulation step and using the remaining time for solving the dispatch instance. In our experiments, we observed that increasing the time for solving simulation instances or increasing the number of simulations did not help improve our solutions. Moreover, we observed that we only need at most 30 seconds to obtain high-quality solutions for the dispatch instance.

We therefore decided to apply the simulation method over and over again. We refer to each application of the simulation method as *simulation cycle*. The first simulation cycle is described in the previous paragraph. In the next simulation cycles, we apply the simulation procedure on the dispatch instance that resulted from the previous cycle. We repeat this N^{cycle} times to obtain the "final" dispatch instance.

Always postpone routes if possible Once we have obtained the dispatch instance, we use our static solver to obtain a solution S_t. We then postpone all routes that do not contain any must-dispatch request. In other words, we remove all routes without must-dispatch requests from S_t to get a new solution S_t^+, which is submitted to the environment instead. This simple postponement strategy already led to significant improvements over the greedy baseline strategy, and also works well in combination with our simulation method.

Parameter choices and tuning The dynamic method introduces multiple new parameters that need to be tuned. We opted for a prescriptive approach identical to the tuning of the static problem. We chose the following three parameter groups: (1) the static solver parameters for solving simulation instances (0.5 seconds time limit), (2) the static solver parameters for solving dispatch instances (30 seconds time limit), and (3) the parameters specific to our dynamic method.

4 Conclusion

With the right implementation and parameter settings, our methods achieved very good results on the two problem variants of the Euro meets NeurIPS vehicle routing challenge.

Acknowledgments and Disclosure of Funding

We are grateful to Marjolein Aerts-Veenstra, Sandjai Bhulai, and Arpan Rijal for their excellent supervision.

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," 2022.

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

[4] Y. Nagata and S. Kobayashi, "A memetic algorithm for the pickup and delivery problem with time windows using selective route exchange crossover," in *Parallel Problem Solving from Nature, PPSN XI* (R. Schaefer, C. Cotta, J. Kołodziej, and G. Rudolph, eds.), Lecture Notes in Computer Science, (Berlin, Heidelberg), pp. 536—545, Springer, 2010.

[5] E. Taillard, "Parallel iterative search methods for vehicle routing problems," *Networks*, vol. 23, no. 8, pp. 661–673, 1993.

[6] L. Accorsi and D. Vigo, "A fast and scalable heuristic for the solution of large-scale capacitated vehicle routing problems," *Transportation Science*, vol. 55, no. 4, pp. 832—856, 2021.

[7] M. Lindauer, K. Eggensperger, M. Feurer, A. Biedenkapp, D. Deng, C. Benjamins, T. Ruhkopf, R. Sass, and F. Hutter, "SMAC3: A versatile Bayesian optimization package for hyperparameter optimization," *Journal of Machine Learning Research*, vol. 23, no. 54, pp. 1–9, 2022.