Hybrid Genetic Search Improvements for the EURO Meets NeurIPS 2022 Vehicle Routing Competition

Brandon M. Reese
SAS Institute
Cary, NC 27513
`Brandon.Reese@sas.com`

Steve Harenberg
SAS Institute
Cary, NC 27513
`Steven.Harenberg@sas.com`

Laszlo Ladanyi
SAS Institute
Cary, NC 27513
`Laci.Ladanyi@sas.com`

Rob Pratt
SAS Institute
Cary, NC 27513
`Rob.Pratt@sas.com`

Yan Xu
SAS Institute
Cary, NC 27513
`Yan.Xu@sas.com`

Abstract

This paper summarizes the finals-qualifying entry from the team, *Miles To Go Before We Sleep*, in the Euro Meets NeurIPS 2022 Vehicle Routing Competition. We outline several aspects of our methodology and the benchmarking strategy that allowed us to measure and track solver improvements. The solver that we submitted to the final round extends the competition's baseline implementation by leveraging offline optimization computations, improving computational performance, and integrating new heuristics.

1 Introduction

In this paper, the authors discuss the high-level strategic and technical methods of our team, *Miles To Go Before We Sleep*. This team produced a competition entry that qualified for the final round of the Euro Meets NeurIPS 2022 Vehicle Routing Competition [1]. The competition task was the construction of a solver for static and dynamic variants of the vehicle routing problem with time windows [2]. The organizers provided a baseline implementation that included a state-of-the-art solver for the static variant but only basic heuristic approaches for the dynamic variant. The scoring system equally incentivized attaining a good rank on each problem variant. Our team focused first on improving the static solver. We reasoned that improvement of the already-competitive static solver would be difficult, so even if the improvements on the static variant were slight, they could yield a large improvement to our leaderboard ranking. We also reasoned that any improvement to the static solver would likewise improve our dynamic solver because the dynamic solver implementation calls the static solver at each epoch. After we were able to consistently achieve a competitive rank on the static variant, we shifted our focus to improving the dynamic solver.

Two keys to making incremental improvements to our solvers were benchmarking and versioning. As we experimented, keeping a consistent method for measuring performance against the public problem instances enabled us to objectively decide whether or not to include experimental features in our final solver. Utilizing separate git branches for each experimental idea enabled us to test our code changes in parallel, merging them back to the main branch only when we were convinced of their efficacy. In Section 2, we describe some details of the methods we explored to improve our solver from the baseline starting point to a final submission that could qualify for the finals.

2 Approach

Our approach to the competition included a number of methodical attempts. Due to time constraints, many of these attempts were explored only viscerally, and much of the work consisted of experimentation that we carried out in parallel. Some attempts produced some improvement, were included in our final submission, and are described in Subsection 2.1. Other attempts did not yield measurable improvements but are noted in Subsection 2.2

2.1 Final approaches

Subsection 2.1.1 describes a technique that improved our benchmarking methods. The other methods described in this section resulted in code changes that were incorporated into our final solver.

2.1.1 Creating subsets of benchmark instances

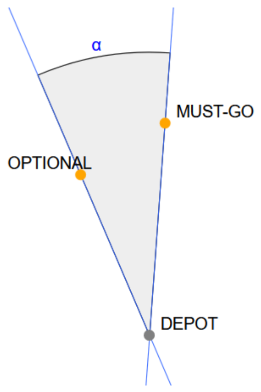
In early prototyping, the time required to evaluate a proposed change against all 250 public instances was often prohibitively slow. In order to iterate more rapidly, we selected smaller subsets to enable faster validation. Instead of choosing these subsets arbitrarily, we selected diverse and representative subsets by solving a maximum dispersion problem. We defined a feature vector for each instance and defined the distances between instances as the Euclidean distance in the feature vector space. Instance features included the number of clients, the vehicle capacity, and the ratio of mean client demand to vehicle capacity. Other derived features included the coefficient of variation of per-client quantities such as demand, distance to the depot, time window duration, and service duration. This method of instance subset selection ensured that solvers that performed well on the subset were likely to generalize well to the full set of instances.

2.1.2 Achieving determinism

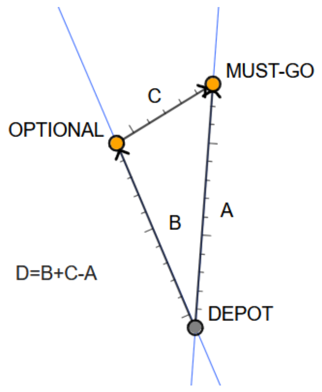
Early in the competition, we noticed that repeated submissions often resulted in significantly different solution costs. Until we addressed the nondeterminism in the solver, our benchmarking efforts would be less effective. We found two causes for non-repeatable solution costs: run-time variance and undefined behavior in the HGS-VRPTW code. The former had a smaller effect and was generally outside of our control in the competition submission environment. The latter was more disruptive and often materialized as run-to-run alternation between two or more execution states with large potential differences in solution cost. We corrected some problematic bugs, including integer overflow, division by zero, and looping in an order-sensitive way over an `unordered_set`. After correcting the potentially undefined behavior, we were able to generate the same sequence of solutions each time we executed the code. This gave us confidence that the performance improvements we obtained were not merely artifacts of nondeterministic execution.

2.1.3 Improving computational performance

Another early emphasis was performance improvement. We found that HGS-VRPTW was extremely well tuned, and we had difficulty improving static scores at first, because our local improvements on the public instances were not generalizing to the hidden instances. Then, we realized that pure performance improvements could enable HGS-VRPTW to get guaranteed equal or better scores on any instance, as long as they did not change the logical flow of the code. We achieved performance improvement in multiple ways. Profiling the code indicated that recursive calls were not properly inlined by the compiler. Rewriting to avoid recursion and force inlining produced an instant 5–10% improvement in execution speed. Next, we attained improvement by swapping data structures. In some cases, we were able to remove $\mathcal{O}(n)$ search loops and instead use $\mathcal{O}(1)$ amortized lookup operations. On top of that, we found that switching the Matrix class from 1D to 2D array implementation (with special care to make sure we still used contiguous memory space) produced a 10% speedup on the `get()` function, which is called very frequently by the solver. We also made improvements to the Python wrapper. The baseline code passed a conservative time limit to HGS-VRPTW to ensure adequate stopping time. We found that by adding the timeout Linux command, we could kill HGS-VRPTW precisely at the desired time with no need for early stopping.



(a) Angle-based



(b) Distance-based

Figure 1: Two types of dispatch heuristic

2.1.4 Applying dispatch heuristics

One effective method for improving solution costs on the dynamic problem variant was the introduction of simple heuristics for client dispatch selection. Rather than dispatching all optional clients (the greedy strategy), we dispatched a subset of the optional clients. We selected that subset by choosing the clients with the lowest score computed by either of two heuristics.

Figure 1a illustrates the angle-based heuristic. For this heuristic, we set the score for each optional client by computing the smallest angle α measured relative to the depot and each must-go client. Figure 1b illustrates the distance-based heuristic. For this heuristic, we set the score for each optional client by computing the smallest insertion distance D measured relative to the depot and each must-go client. As for the size of the dispatched subset, we tried three tunable options: a fixed threshold for α or D, a fixed proportion of all optional clients, or a fixed multiple of the number of must-go clients. In the corner case of zero must-go clients, we dispatched no optional clients.

Additionally, one simple postprocessing rule also proved effective. If any route computed by HGS-VRPTW contained only optional clients, we removed that route and deferred those clients to a later epoch.

2.1.5 Applying scheduled dispatch thresholds

Toward the end of the qualification phase, we experimented with scheduled (rather than fixed) thresholds. We hypothesized that it would be effective to use a more aggressive threshold when fewer epochs remained. We confirmed this by using data generated by massive simulations of various threshold schedules.

2.2 Other explored approaches

Each of the methods described below, although promising, resulted in either insufficient benchmark evidence of improvement or poor generalization to the hidden qualification instances. They were not incorporated into our final solver.

2.2.1 Preprocessing based on time windows

Upon inspecting the provided public instances, we noticed that many links were not feasible because $T_{0s} + D_s + C_{st} > T_{1t}$, where T_{0s} is the earliest allowed arrival at the source, D_s is the service duration at the source, C_{st} is the travel time from the source to the target, and T_{1t} is the latest allowed arrival at the target. As a preprocessing step, we attempted to discourage the solver from utilizing these links by setting their costs to a large value.

2.2.2 Searching the HGS-VRPTW tuning parameter space

The HGS-VRPTW solver includes many tunable parameters. In an attempt to select sets of parameters that performed better than the default parameters, we performed hyperparameter optimization by using the Optuna Python module [3]. To make the most effective use of computing resources, we implemented a multiprocessing wrapper around Optuna. Although we evaluated more than 10,000 parameter combinations over the public benchmark instances and submitted solvers that used the best-performing tuning parameter configurations, none of the submitted tuning configurations yielded improvement on the hidden qualification instances.

2.2.3 Warm starting and VROOM

We extended HGS-VRPTW by implementing warm start functionality. In this mode, one or more feasible solutions could be passed as inputs, and these solutions would be included in the construction of the initial population. We hypothesized that including good solutions in the initial population would enable HGS-VRPTW to eventually converge to better final solutions; however, the experiments to test this hypothesis were inconclusive. We also incorporated the open-source VROOM[1] solver by writing Python wrapper code. This wrapper enabled the interchangeable execution of VROOM before, after, or instead of HGS-VRPTW. Whether running the VROOM solver alone, VROOM warm-started by HGS-VRPTW, or HGS-VRPTW warm-started by VROOM, we did not find a combination that consistently provided better results than HGS-VRPTW alone.

2.2.4 Solving VRPTW subproblems

One promising approach to find good VRPTW solutions faster is subproblem decomposition [4]. In our implementation, we defined subproblems from an existing feasible solution. First we represented each route by its centroid coordinate. Then we bundled each route with its k-nearest neighbor routes according to centroid position. The union of clients belonging to bundled routes defined the set of clients in the subproblem. We used heuristics similar to those reported in [4] to select the next subproblem, employed HGS-VRPTW to solve the subproblem, and then iterated that process until time expired. This method showed promise on local benchmarks, yielding a 0.4% (dynamic variant) score improvement when considering all 250 instances and averaged over five instance seeds. However, each attempt to submit the solver from this code branch failed on the hidden instances with an apparent crash that was difficult to diagnose without error logs.

2.2.5 Clustering for dispatch selection

We attempted to improve client dispatch selection by using multiple clustering methods. We employed the following basic procedure. First, break the set of optional and must-go clients into clusters such that clustered clients were near in proximity. Then, dispatch optional clients only if they belong to the same cluster as at least one must-go client. We tried various clustering methods from the scikit-learn [5] clustering module, and we also ran VROOM with a limited number of vehicles and client priorities, treating the sets of nodes in solution routes as clusters. In our experimentation, we found that these clustering approaches yielded results that were better than the greedy baseline solver but not better than the simple heuristic selection methods described in Subsection 2.1.4.

2.2.6 Machine learning for dispatch threshold selection

After observing the benefit of supplying a schedule of thresholds, as described in Subsection 2.1.5, we experimented with building machine learning models to identify the best threshold for a given epoch in light of what we know about the instance (namely, its static properties). For this task, we used the same feature set described in Subsection 2.1.1 and generated training data by solving the dynamic problem for many combinations of threshold, instance, and instance seed. We tried several regression methods from scikit-learn, but ultimately, the models lacked predictive power. The main difficulty we observed was that the optimal choice of threshold depended much more heavily on the choice of instance seed than on the choice of instance. Because of this, when we averaged our results over instance seed, the observed optimal threshold was uncorrelated with the set of instance features.

[1]http://vroom-project.org/

3 Conclusion

In summary, we have explored a number of methodologies in the construction of our competition entry. Although these methods varied in sophistication, the decision of whether to include each method ultimately depended on performance across the public benchmark and private qualification instances. In general, simpler methods were successful because they are less susceptible to overfitting based on the limited set of public instances.

References

[1] W. Kool and L. Bliek, "EURO meets NeurIPS 2022 vehicle routing competition," 2022.

[2] W. Kool, B. Laurens, 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.

[3] T. Akiba, S. Sano, T. Yanase, T. Ohta, and M. Koyama, "Optuna: A next-generation hyperparameter optimization framework," in *Proceedings of the 25rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2019.

[4] S. Li, Z. Yan, and C. Wu, "Learning to delegate for large-scale vehicle routing," in *Advances in Neural Information Processing Systems* (A. Beygelzimer, Y. Dauphin, P. Liang, and J. W. Vaughan, eds.), 2021.

[5] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.