Local Search with Learned Constraints for Last Mile Routing

William Cook

Combinatorics and Optimization, University of Waterloo

200 University Avenue West, Waterloo, ON, Canada

Stephan Held

Research Institute for Discrete Mathematics, University of Bonn

Lennéstr. 2, 53113 Bonn, Germany

Keld Helsgaun

Computer Science, Roskilde University

P.O. Box 260, DK-4000, Roskilde, Denmark

Abstract

We describe our submission to the Amazon Last Mile Routing Research Challenge. The optimization method we employ utilizes a simple and efficient penalty-based local-search algorithm, first developed by Helsgaun to extend the LKH traveling salesman problem code to general vehicle-routing models. We further develop his technique to handle combinations of routing constraints that are learned from an analysis of historical data. On a target set of 1,107 training instances, our submitted code achieves a mean score of 0.01989 and a median score of 0.00752. The simplicity of the method may make it suitable for applications where machine learning can discover rules that are expected (or desired) in high-quality solutions.

1. Introduction

Routing in the Amazon Last Mile Routing Research Challenge is at the level of an individual driver. The data provide a list of stops a driver must visit to deliver packages on a given day, together with point-to-point travel times. The travel times are asymmetric, that is, the time to travel from stop x to stop y may not be the same as the time to travel from y back to x. The task of finding the quickest route for the driver is an instance of the asymmetric traveling salesman problem (ATSP).

Direct optimization of routes is not the focus of the challenge, however. Teams are asked to match, as best they can, the sequences of stops in actual routes driven by van drivers, who may have taken into consideration warehouse operations, current road conditions, types and sizes of packages to be delivered, and other factors. In machine-learning fashion, models and algorithms developed with the help of provided training data are evaluated by their performance on an additional large set of routes that are kept hidden from competing teams.

1.1. Initial scores

As a first step towards a solution procedure, we consider optimal tours for the ATSP instances. The Concorde code (Applegate et al. (2006)) solves symmetric instances of the TSP, so we applied a standard transformation of the ATSP to the TSP, splitting each stop into a pair of nodes, one for incoming edges and the other for outgoing edges. The TSP instances thus obtained were solved with Concorde in an average of 142 seconds of computation time on a single core of a linux workstation, equipped with an Intel Xeon Gold 6238 CPU @ 2.10GHz processor. This is not an efficient way to solve instances of the ATSP, but it allowed us to easily obtain optimal tours for the set of 1,107 High+Delivered training instances (those with driver routes that are classified as 'High' quality and have no undelivered packages; the target class for the final competition).

Solution Set	Mean Travel Time	Score
Driver Tours	12250.0 sec	0.0
ATSP Tours	10853.3 sec	0.07030

The driver tours are on average 12.9% longer than routes that minimize travel time.

The optimal ATSP tours achieved a competition score of 0.07030. The score of a tour is a measure of its similarity with the corresponding driver tour. Details of the score computation can be found at the challenge GitHub site `https://github.com/MIT-CAVE/rc-cli/tree/main/scoring`. A lower score indicates a better match, the driver tour itself scoring a perfect 0.0.

The ATSP tours are only a weak approximation of the driver sequences. For example, the tour order does not reflect any conditions related to the warehouse operation or to the driver's storage of packages in the van he/she will be using that day. Regarding this, the training data provide a zone ID for each stop, indicating a possible clustering of the route's stops in a tour.

There are an average of 20 distinct zones represented in each of the High+Delivered routes, giving an average of approximately 7 stops per zone. The zones partition the set of nodes in the ATSP instance, where we create a special zone containing only the delivery station. (In cases where a stop in the training data does not have a zone ID, we assign to it the ID of the closest stop (in Euclidean distance) associated with the same station.) It is not difficult to modify an ATSP instance to force the stops in each of these zones to appear consecutively in any optimal tour. This can be achieved by adding a large constant M to the travel time of any edge joining stops in distinct zones. With this modification, we again used the ATSP to TSP transformation and solved the resulting instances with Concorde.

Solution Set	Mean Travel Time	Score
Clustered-ATSP Tours	11235.2 sec	0.04866

The optimal clustered-ATSP tours obtained were 3.5% longer than the tours without the cluster restrictions, but they reduced the similarity score to 0.04866.

1.2. Precedence constraints, tour neighbors, and super clusters

We view the clustered instances as the starting point for the Last Mile Challenge. Our research program aimed to obtain properties of driver tours that could restrict the clustered-ATSP instances to achieve a better match with the choices made by the driver and by the warehouse operators. The restrictions are obtained from analysis of the routes in the training data, making direct use of the zone-to-zone sequences in the driver tours, as well as the zone ID patterns these sequences create.

All new restrictions we consider are at the inter-zone level, while maintaining the clustering of stops within each zone. A natural constraint is to force a precedence between a pair of zones (a, b) as they appear in a tour, that is, if we let visit(x) denote the position of zone x in the tour (with the delivery-station zone having position 0), then a *precedence constraint* for (a, b) requires visit$(a) <$ visit(b). A *path constraint* for (a, b) is the stronger restriction that zone a immediately precedes zone b in the tour, that is, visit$(a) =$ visit$(b) - 1$. We also consider *neighbor constraints*, requiring that zone a is either immediately before or immediately after zone b in the tour, that is, $|\text{visit}(a) - \text{visit}(b)| = 1$. Furthermore, we permit in our models logical combinations of these three types of constraints; particularly useful are binary disjunctions, stating that for two specified constraints (A, B), a solution tour must satisfy either A or B.

Together with constraints involving pairs of zones, we also create super clusters, forming a partition of the zone clusters, and super-super clusters, forming a partition of the super clusters. This hierarchy allows us to create precedence, path, and neighbor constraints for pairs of super clusters and pairs of super-super clusters, further guiding the algorithm-produced tours towards sequences followed by actual drivers.

1.3. Optimization engine

Super clusters, super-super clusters, and neighbor constraints (at all levels of the hierarchy) can be incorporated into an ATSP model by modifying travel costs, as we described in the clustered-ATSP case. The inclusion of precedence constraints, path constraints, and logical disjunctions make such a transformation much more difficult. We therefore do not rely on an ATSP solver in our work, developing instead a penalty-based local-search heuristic that can handle multi-level clustering and logical combinations of our three types of constraints. The new heuristic optimizer allows us to compute high-quality tours in computation time well under the limit set in the final competition.

2. Literature Review

The local-search paradigm is simple: starting with any candidate solution to a problem, repeatedly look for small alterations that can possibly lead to a better solution. The idea dates back to work on the TSP, starting with the 2-opt heuristic by Flood (1956), that was quickly extended to 3-opt by Bock (1958) and Croes (1958), and later to a variable k-opt algorithm by Lin & Kernighan (1973). Extensions of the basic paradigm are discussed in surveys by Lourenço et al. (2003) (iterated local search), Hoos & Stützle (2005)

(stochastic local search), and Alsheddy et al. (2018) (guided local search), and in the book Aarts & Lenstra (2003), covering a range of applications in discrete optimization.

Of particular interest in our work is the use of local search in the area of constraint programming, where penalties are used to guide the heuristic towards feasible solutions; see Hoos & Tsang (2006) for a survey. Ideas from this constraint-programming work were adopted to handle time-window-constrained routing problems in López-Ibáñez & Blum (2010) and Nagata et al. (2010).

Helsgaun (2017) incorporated penalty-based search into his LKH-3 code for vehicle-routing models, greatly extending the reach of his well-known TSP code LKH (Helsgaun (2000)). We use in our work the simple and efficient method deployed in LKH-3, to measure simultaneously the length of the route and the penalties incurred when constraints are violated.

3. Methodology

Our approach adopts modest machine-learning techniques, but combines this with a powerful heuristic optimization method to exploit the discovered properties of driver routes.

3.1. LKH-AMZ

The tour-finding heuristic is called LKH-AMZ, acknowledging its origins in the LKH-3 code. LKH-AMZ streamlines LKH-3 by implementing a restricted set of k-opt moves, suitable for the short computation times called for in the Last Mile Challenge. At the same time, LKH-AMZ expands the family of constraints that can be handled, giving us great flexibility in adopting rules that are learned from training routes.

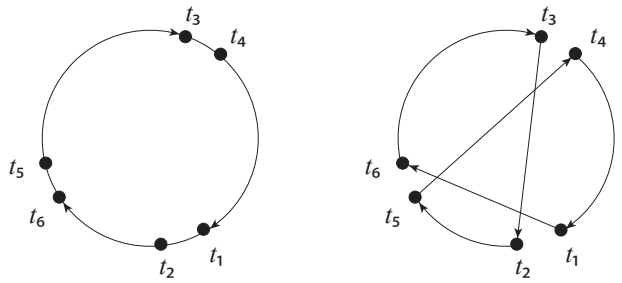
3.1.1. Penalizing constraint violations

LKH-AMZ treats the initial zone-clustering restrictions as hard constraints, adopting the travel-cost transformation described in Section 1.1. The remaining restrictions are treated as soft constraints, handled by penalizing any violations.

Cluster, super-cluster, and super-super-cluster penalties are evaluated with a simple $O(n)$ mechanism that loops through the stops in tour order, counting the numbers e_c, e_{sc}, and e_{ssc} of tour edges entering different clusters, super clusters and super-super clusters, respectively. If we have k_c clusters, k_{sc} super clusters, and k_{ssc} super-super clusters, then the assigned penalties are $10 * (e_c - k_c)$, $10 * (e_{sc} - k_{sc})$ and $10 * (e_{ssc} - k_{ssc})$, respectively. Note that cluster penalties are not strictly necessary, since they are handled by the cost transformation.

The precedence, path, and neighbor constraints are evaluated by examining the tour positions of the constrained pairs of clusters, super clusters, and super-super clusters. It is possible to set the penalties at different values for individual pairs, but to avoid over-fitting we adopt only two penalty levels. Cluster-precedence constraints and cluster-neighbor constraints receive a penalty of 1 when violated, and cluster-path constraints and all super-cluster and super-super-cluster constraints receive the penalty 1000.

Disjunctions of constraints are evaluated as the minimum penalty of each pair in the disjunction.



(a) Choice of points. (b) Result of 3-opt move.

Figure 1: Special 3-opt move.

3.1.2. Tour representation

We make use of the standard ATSP to TSP transformation (see Jonker & Volgenant (1983)). With each node i of the original problem there is associated a dummy node $i + n$ and a fixed edge $(i, i + n)$ having travel cost 0. LKH-AMZ organizes its tours so that every original node follows its dummy node in the tour orientation. A two-level list data structure (see Fredman et al. (1995)) is used to represent the tour, supporting fast k-opt operations.

3.1.3. Special 3-opt, 4-opt moves

We say a tour T' *improves* a tour T if at least one of its penalty or length is strictly less than that of T and neither of these values has increased. Given a tour T, and a non-fixed edge (t_1, t_2), the function SpecialMove attempts to find a 3-opt or 4-opt move that improves T.

Since every original node must follow its dummy node in the tour, a move is not allowed to reverse any segment of the tour. This leaves only two possible 3-opt and 4-opt moves.

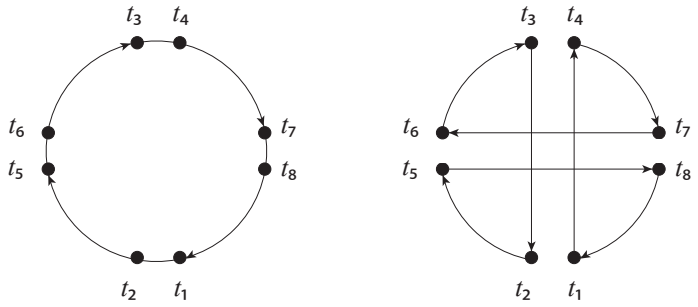
Figure 1a shows the choice of nodes t_1, t_2, t_3, t_4, t_5, and t_6 for a 3-opt move. Note that node t_5 must lie between t_2 and t_3 in the tour's orientation. Figure 1b shows the result of the move. Observe that no segments have been reversed.

Figure 2a shows the choice of nodes $t_1, t_2, t_3, t_4, t_5, t_6, t_7$, and t_8 for a 4-opt move. Note that again t_5 must lie between t_2 and t_3, and t_7 must be between t_4 and t_1. The resulting move is displayed in Figure 2b. To keep the time complexity low, only four possible combinations of (t_5, t_6) and (t_7, t_8) are tried (those nearest to t_1, t_2 and t_3, t_4, respectively).

After a move $T \Rightarrow T'$ has been made, it is tested whether T' improves T. In that case, the current tour T is set to T'. Otherwise, the move is retracted.

3.1.4. Candidate edges

LKH-AMZ restricts the search for moves by means of candidate edges. These are the edges that are considered for possible inclusion in the tour. To reduce the number of such edges, an edge-elimination procedure based on minimum spanning trees and subgradient optimization is applied (Helsgaun (2000)).



(a) Choice of points. (b) Result of 4-opt move.

Figure 2: Special 4-opt move.

The candidate edge set is further sparsified by eliminating those edges whose inclusion would violate any neighbor constraints. This latter step both speeds up the search for moves and guides the search towards solutions satisfying the constraints.

3.1.5. Iterated local search

A *trial* in LKH-AMZ is a sequence of 3-opt and 4-opt moves found by SpecialMove, where the search for these moves is terminated according to rules that trade off the speed of the computation with the likelihood of finding further improvements. (See Applegate et al. (2006) for a detailed discussion in the context of the pure TSP.) In each *run* of the code, n trials are executed, where n is the number of nodes in the input data.

The trials are organized as an iterated local search. The idea is to allow the code to quickly sample local minima in the neighborhood of the best tour T^* that has been found thus far. The initial trial begins with a pseudo-random tour. In each subsequent trial, the current T^* is "kicked" by performing a 4-opt move of the type indicated in Figure 2; the 4-opt kick is selected using the Rohe long-edge rule with a 50-step random walk, as described in Applegate et al. (2003). The kicked tour is re-optimized via a sequence of calls to SpecialMove. If this search produces a tour T that improves T^*, then we replace T^* by T.

3.1.6. Multiple runs

Until a specified time limit is reached, we carry out multiple runs of the full iterated local search. Each run is initiated with a pseudo-random tour. We evaluate the resulting tour T by the function $v(T) = PenaltyMultiplier * Penalty(T) + Cost(T)$, and record T as a new best tour if $v(T)$ is less than the $v(\cdot)$ value of our current best tour. The role of the scaling factor $PenaltyMultiplier$ is to allow us to have both the penalty and cost of the tour play a role in the decision to accept the result of the run; it is set to 1500 in our tests, bringing the typical scaled $Penalty(T)$ to a value comparable to $Cost(T)$.

3.2. Zone precedence constraints from training data

To predict the order in which a driver traverses the zones in a new routing instance r, we compute a reference route $q^\star$ from the training instances. We choose $q^\star$ among all routes that start at the station of r,

sharing the maximum (scaled) number of common zones with r. As we favor reference routes with a 'High' route score, we scale the number of common zones by a factor 2/1.5/1 for 'High'/'Medium'/'Low' scored routes. We then extract zone-order constraints based on $q^\star$ as follows.

During the model-build phase we compute a *zone graph* for every route. It contains a vertex for every visited zone and an edge (a, b) if the driver served a package in zone a immediately before a package in zone b. Contracting all its strongly connected components yields the *component path*. Its vertices consist of strongly connected components of zones. (See Tarjan (1972).)

In the apply phase, we create a precedence constraint $\text{visit}(a) < \text{visit}(b)$ for each pair (a, b) of zones whose strongly connected components $A \ni a$ and $B \ni b$ are joined by an edge (A, B) in the component path of the reference route.

Adding precedence constraints to the Clustered-ATSP instances reduces the score from 0.04866 to 0.03414, where the tours are obtained by running LKH-AMZ for 10 seconds per instance.

Alternatively, we use the *transitive closure* of all such precedence constraints. This reduces the score to 0.03157. However, in combination with constraints from Section 3.3, non-transitive constraints gave better results on average.

We also tried to extract zone-order constraints from multiple reference routes, but did not find a model that improves the single reference route approach.

3.3. Cluster rules from zone IDs

The zone IDs assigned to stops have the form Γ-$x.y\Delta$, where Γ and Δ are capital letters and x and y are integers. In Table 1 we list these IDs in the order they appear in the driver tours for each of the first four High+Delivered routes. The lists have patterns, suggesting drivers are following higher-level clusters created by transitions in the four components of the zone IDs. In our build phase, we create such a clustering by selecting a subset $\mathcal{S}$ of the symbols $\{\Gamma, x, y, \Delta\}$ and grouping all zone IDs that have matching values in the $\mathcal{S}$ positions. Super clusters are determined by a selection $\mathcal{S}$ of three symbols, super-super clusters are determined by a selection $\mathcal{T} \subset \mathcal{S}$ of two symbols, and a top-level clustering is determined by $\mathcal{U} \subset \mathcal{T}$ having a single symbol. For example, setting $\mathcal{S} = \{\Gamma, x, \Delta\}$, $\mathcal{T} = \{\Gamma, x\}$, and $\mathcal{U} = \{\Gamma\}$ gives for route amz0002 from Table 1 the super clusters {A-2.2E, A-2.1E}, {A-2.1D, A-2.2D, A-2.3D}, {A-2.3C, A-2.2C, A-2.1C}, {A-2.1B, A-2.2B} and a single super-super cluster consisting of all of the zones. In our submission, the choices of $\mathcal{S}$, $\mathcal{T}$, and $\mathcal{U}$ are made in a build-phase computation, minimizing the number of times the training tours cross the components of the partitions of zones at each clustering level.

Adding these super clusters and super-super clusters to our test instances improves the score to 0.02347 with 10-second runs of LKH-AMZ. Going further with the analysis, within each super cluster C we sort by zone ID the clusters contained in C, and add neighbor constraints for each adjacent pair of clusters in the sorted order. These constraints put the clusters within a super cluster in either sorted or reverse sorted order. To attempt to force our tours to match the driver's choice between these two orderings, for neighboring pairs

Table 1: Sequences of zone IDs in driver tours

amz0002	amz0003	amz0012	amz0019
Station	Station	Station	Station
A-2.2E	B-11.1G	A-2.1E	M-10.3D
A-2.1E	B-11.1H	A-2.2D	M-10.3C
A-2.1D	B-11.2H	A-2.3D	M-10.2C
A-2.2D	B-12.1G	A-2.3C	M-10.1C
A-2.3D	B-12.3G	A-2.2C	M-10.1B
A-2.3C	B-12.2G	A-2.1D	M-10.2B
A-2.2C	B-12.3G	A-2.1C	M-10.3B
A-2.1C	B-12.1H	A-2.1B	M-10.3A
A-2.1B	B-12.2H	A-3.2A	M-10.2A
A-2.2B	B-12.3H	A-2.1A	M-10.1A
	B-12.3J	A-2.2B	M-10.2A
	B-12.2J	A-2.3B	P-10.1A
	B-12.1J	A-2.3A	P-10.2A
	B-11.1J	A-3.1A	P-10.3A
	B-11.2J	A-2.2A	P-10.3B
	B-11.3J	A-2.3A	P-10.2B
	B-11.3H		P-10.1B
	B-11.3J		P-10.1C
	B-11.2G		P-10.2C
			P-10.3C
			P-10.3D

of super clusters (C, D) we possibly add zone-level neighbor constraints for pairs of zones (c, d), with c being either the first or last zone in the sorted order for C and d being either the first or last zone in the sorted order for D. The neighbor constraint (c, d) is added if these zones have matching values for the unique symbol u in $\{\Gamma, x, y, \Delta\} \setminus \mathcal{S}$. If there are two such pairs of zones for (C, D), then we create a disjunction for the pair of neighbor constraints. Corresponding constraints are also added for sorted orders of super clusters within super-super clusters, and for sorted orders of super-super clusters within each top-level cluster. Adding the entire collection of new constraints leads to a score of 0.02146.

To complement the constraints created with zone ID patterns, we also add super-cluster path constraints derived from the set of training routes. Here we follow the idea from our work on cluster-level precedence. For a routing instance r, we find a reference route $q^\star$ having the greatest number of super clusters in common with r. We add a path constraint for pairs of super clusters (C, D), such that C and D appear in both r and $q^\star$, the clusters C and D are each entered exactly twice in $q^\star$, and C and D appear consecutively in $q^\star$. These additional constraints improve the score to 0.02030 and we adopt them in our default code.

As an alternative, it is possible to add a super-cluster precedence constraint for (C, D) rather than a path constraint, but this gives the slightly worse score of 0.02079.

3.4. Merging families of tours

LKH-AMZ is designed to very quickly find good tours satisfying all or most of the specified constraints. Indeed, it is not always productive to increase the LKH-AMZ time limit.

LKH-AMZ Time per Route	10 sec	20 sec	30 sec	40 sec	50 sec	60 sec
Score	0.02030	0.02001	0.01997	0.01997	0.02003	0.02002

We instead use a portion of the computation time to run LKH-AMZ on an alternate set of constraints and then merge the two collections of tours. To merge, we compute the travel times t_d and t_a for a route's default and alternate tours, then select the default if $t_d \leq MergeFactor * t_a$, and otherwise select the alternate. Setting $MergeFactor = 1.01$ allows us to replace outlier tours in our collection.

In our submission, the alternate set of constraints is obtained from the default set by selecting instead the transitive-closure precedence constraints in Section 3.2 and the super-cluster precedence constraints in Section 3.3; we allocate twice as much running time for the default constraints.

Time per Route	Default	Alternate	Merged
10 sec + 5 sec	0.02030	0.02176	0.02009
20 sec + 10 sec	0.02001	0.02116	0.01989
40 sec + 20 sec	0.01997	0.02113	0.01989

3.5. Computer implementation

The LKH-AMZ heuristic solver, written in the C programming language, consists of approximately 5,300 lines of code. Python scripts, totaling 2,600 lines, are used for analyzing zone information, extracting constraints, and visualizing tours. An additional 500 lines of C implement an internal scoring function, 300 lines of shell scripts control the LKH-AMZ solver and execute the merge routine, and 100 lines of scripts control the build and apply phases of the submission to the challenge. All codes and scripts are available under the MIT License at `https://github.com/heldstephan/jpt-amz`.

4. Results and Conclusions

Codes submitted to the Last Mile Challenge are evaluated on a set of 3,050 routes, running on an AWS EC2 m5.4xlarge server and a time limit of 12 hours for the build phase and 4 hours for the apply phase. The m5 server has an 8-core processor and supports 16 virtual cores. The results presented above were carried out on a linux server equipped with two 16-core Intel Xeon Gold 5218 CPU @ 2.30GHz processors, supporting a total of 64 virtual cores. The two machines have roughly similar singe-core performance.

Our build-phase analysis completes in under 5 minutes on a singe core of the linux server, so well within the competition time limit. To accommodate the 20s+10s merge runs in the 4-hour apply-phase limit, we execute in parallel the LKH-AMZ runs on individual routes. Using 64 threads on the linux server, the total wall-clock computation time for the 20s+10s merge was 617 seconds.

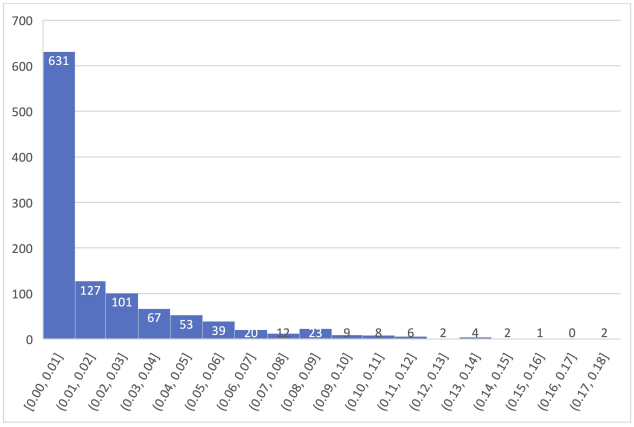


Figure 3: Histogram of scores for the 1,107 High+Delivered instances.

The `model_apply` script in our submitted code determines the amount of time to allocate to LKH-AMZ after the constrained ATSP instances are created, aiming for a total run time of 3.5 hours using 16 threads. In a test on an AWS EC2 m5.4xlarge server using 3,050 routes, the code adopted a 41s+20s merge.

4.1. Distribution of scores

We have thus far reported only the mean value of the 1,107 individual tour scores for the High+Delivered routes, which is the measure used to rank the submissions in the Last Mile Challenge. More detail can be seen in the histogram of the scores given in Figure 3. The mean score is greatly impacted by a small number of poor results. Indeed, the median score of 0.00752 is well under the 0.01989 mean.

4.2. Final remarks

- We were not successful in learning useful constraints for individual stops, due in part to the sparse coverage of the training routes when observed at the stop level. We expect that more sophisticated learning approaches developed by other teams could be combined with our optimization engine.
- LKH-AMZ can very quickly, in a small number of seconds, obtain low travel-time tours that achieve good competition scores. This speed may make the code suitable for real-time optimization, where updated travel times could be uploaded and a new tour computed.
- LKH-AMZ is set to handle time-window constraints, but we did not utilize this in our submission. We found that the small number of such constraints did not impact the scores we obtained.
- The choices for $PenaltyMultiplier$ and $MergeFactor$ can be learned in the build phase, but given the single trial permitted in the final evaluation, we chose the conservative approach of setting their values to 1.01 and 1500 respectively, the midpoints of the [1.00,1.02] and [1000,2000] ranges we observed as acceptable values in our study. A trial run of a build script on an AWS server, covering a random sample of 1,000 of the High+Delivered training instances, selected $PenaltyMultiplier = 1250$ and $MergeFactor = 1.016$, but with only a slight score improvement over the default midpoint values.

References

Aarts, E., & Lenstra, J. K. (2003). *Local Search in Combinatorial Optimization*. Princeton: Princeton University Press.

Alsheddy, A., Voudouris, C., Tsang, E. P. K., & Alhindi, A. (2018). Guided local search. In R. Martí, P. M. Pardalos, & M. G. C. Resende (Eds.), *Handbook of Heuristics* (pp. 261–297). Cham: Springer International Publishing. doi:10.1007/978-3-319-07124-4_2.

Applegate, D., Cook, W., & Rohe, A. (2003). Chained Lin-Kernighan for large traveling salesman problems. *INFORMS Journal on Computing*, *15*, 82–92. doi:10.1287/ijoc.15.1.82.15157.

Applegate, D. L., Bixby, R. E., Chvátal, V., & Cook, W. J. (2006). *The Traveling Salesman Problem: A Computational Study*. Princeton: Princeton University Press.

Bock, F. (1958). *An algorithm for solving "traveling-salesman" and related network optimization problems*. Research Report Armour Research Foundation. Presented at the Operations Research Society of America Fourteenth National Meeting, St. Louis, October 24, 1958.

Croes, G. A. (1958). A method for solving traveling-salesman problems. *Operations Research*, *6*, 791–812. doi:10.1287/opre.6.6.791.

Flood, M. M. (1956). The traveling-salesman problem. *Operations Research*, *4*, 61–75. doi:10.1287/opre.4.1.61.

Fredman, M., Johnson, D., Mcgeoch, L., & Ostheimer, G. (1995). Data structures for traveling salesmen. *Journal of Algorithms*, *18*, 432–479. doi:https://doi.org/10.1006/jagm.1995.1018.

Helsgaun, K. (2000). An effective implementation of the Lin–Kernighan traveling salesman heuristic. *European Journal of Operational Research*, *126*, 106–130. doi:https://doi.org/10.1016/S0377-2217(99)00284-2.

Helsgaun, K. (2017). *An extension of the Lin-Kernighan-Helsgaun TSP solver for constrained traveling salesman and vehicle routing problems*. Technical Report Roskilde Universitet. URL: http://akira.ruc.dk/~keld/research/LKH/LKH-3_REPORT.pdf.

Hoos, H. H., & Stützle, T. (2005). *Stochastic Local Search*. The Morgan Kaufmann Series in Artificial Intelligence. San Francisco: Morgan Kaufmann. doi:https://doi.org/10.1016/B978-155860872-6/50018-4.

Hoos, H. H., & Tsang, E. (2006). Chapter 5 - Local search methods. In F. Rossi, P. van Beek, & T. Walsh (Eds.), *Handbook of Constraint Programming* (pp. 135–167). Elsevier volume 2 of *Foundations of Artificial Intelligence*. doi:https://doi.org/10.1016/S1574-6526(06)80009-X.

Jonker, R., & Volgenant, T. (1983). Transforming asymmetric into symmetric traveling salesman problems. *Operations Research Letters*, *2*, 161–163. doi:https://doi.org/10.1016/0167-6377(83)90048-2.

Lin, S., & Kernighan, B. W. (1973). An effective heuristic algorithm for the traveling-salesman problem. *Operations Research*, *21*, 498–516. doi:10.1287/opre.21.2.498.

López-Ibáñez, M., & Blum, C. (2010). Beam-ACO for the travelling salesman problem with time windows. *Computers and Operations Research*, *37*, 1570–1583. doi:https://doi.org/10.1016/j.cor.2009.11.015.

Lourenço, H. R., Martin, O. C., & Stützle, T. (2003). Iterated local search. In F. Glover, & G. A. Kochenberger (Eds.), *Handbook of Metaheuristics* (pp. 320–353). Boston, MA: Springer US. doi:10.1007/0-306-48056-5_11.

Nagata, Y., Bräysy, O., & Dullaert, W. (2010). A penalty-based edge assembly memetic algorithm for the vehicle routing problem with time windows. *Computers and Operations Research*, *37*, 724–737. doi:https://doi.org/10.1016/j.cor.2009.06.022.

Tarjan, R. (1972). Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, *1*, 146–160. doi:10.1137/0201010.