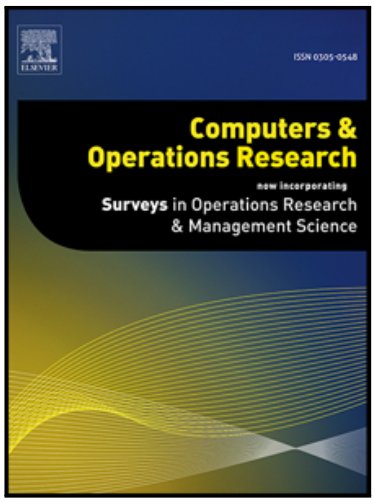
Knowledge-guided local search for the vehicle routing problem



Knowledge-guided local search for the Vehicle Routing Problem

Florian Arnold, Kenneth Sörensen

PII: S0305-0548(19)30002-4
DOI: https://doi.org/10.1016/j.cor.2019.01.002
Reference: CAOR 4624

To appear in: *Computers and Operations Research*

Received date: 26 April 2018
Revised date: 3 October 2018
Accepted date: 7 January 2019

Please cite this article as: Florian Arnold, Kenneth Sörensen, Knowledge-guided local search for the Vehicle Routing Problem, *Computers and Operations Research* (2019), doi: https://doi.org/10.1016/j.cor.2019.01.002

Highlights

- Local search suffices to compute high-quality solutions for routing problems in a short time
- Large neighborhoods combined with a well-implemented pruning make local search effective
- Problem-specific knowledge can help to guide the search more effectively
- The heuristic scales well in problem-size and can be applied to problem variants

Knowledge-guided local search for the Vehicle Routing Problem

Florian Arnold [1a], Kenneth Sörensen[a]

[a] *University of Antwerp, Departement of Engineering Management, ANT/OR - Operations Research Group*

Abstract

Local search has been established as a successful cornerstone to tackle the Vehicle Routing Problem, and is included in many state-of-the-art heuristics. In this paper we aim to demonstrate that a well-implemented local search on its own suffices to create a heuristic that computes high-quality solutions in a short time. To this end we combine three powerful local search techniques, and implement them in an efficient way that minimizes computational effort. We conduct a series of experiments to determine how local search can be effectively combined with perturbation and pruning, and make use of problem-specific knowledge, to guide the search to promising solutions more effectively. The heuristic created in this way not only performs well on many benchmark sets, it is also straightforward in its design and does not contain any components of which the contribution is unclear.

Keywords: vehicle routing problem, local search, heuristics, metaheuristics

1. Introduction

The vehicle routing problem (VRP) is one of the most intensively studied problems in the field of combinatorial optimization. The objective of the VRP is to find a set of vehicle routes that, starting from and ending at a depot, visit a set of customers in such a way that the total cost traveled by all vehicles is minimal. Customers and the depot are nodes of a complete, undirected graph where the cost of an edge (i, j) corresponds to the Euclidean distance $c(i, j)$ between the adjacent nodes. Customers have a known demand, and the total demand served by each vehicle cannot exceed its capacity (where all vehicles are assumed to be identical). For a more formal introduction we refer to Laporte (2007). Notwithstanding its simplicity, the VRP and its many variants have considerable importance in practice and underlie a large number of commercially available logistic planning tools.

Even though significant progress has been made in the development of exact methods for the VRP, state-of-the-art algorithms can only solve relatively small instances of this NP-hard problem to optimality within a reasonable computing time limit. A significant research effort has therefore been devoted to the design of heuristics that attempt to

[1]corresponding author. Email: florian.arnold@uantwerpen.be

find a good solution quickly, but do not guarantee to find the optimal solution in a finite amount of time.

In the last decade many successful heuristics have been developed that can solve instances of several hundred customers to near-optimality in a few minutes of computing time (Vidal et al., 2013a). Most heuristics are based upon metaheuristic designs, and a comprehensive survey about the plethora of heuristic designs is provided by Salhi (2014). Many authors have shown that high-quality results can be achieved with different designs, most notably with genetic algorithms (Vidal et al., 2013a), a mix of exact solvers and heuristics (Subramanian et al., 2013) and memetic algorithms (Nagata and Bräysy, 2009). However, one component that probably all good heuristics for the VRP have in common is local search.

Local search has proven to be the cornerstone of many solution techniques for various combinatorial optimisation problems. Complete metaheuristic designs revolve around an effective local search, e.g., variable neighborhood search (Mladenović and Hansen, 1997) or guided local search (Voudouris and Tsang, 2003), and it has been particularly successful to solve the Traveling Salesman Problem (TSP) (Lin and Kernighan, 1973). In the context of the VRP, many effective local search operators have been developed over the years, and have been condensed in an online library (Groër et al., 2010).

Despite its known success, it appears that most state-of-the-art VRP heuristics use local search to boost a heuristic framework, rather than the other way around. As a consequence, less research has been devoted for a deeper analysis of how succesful local search operators can be setup and implemented effectively on their own. That such an analysis and refinement of local search can result in impressive solution methods has already been shown for the TSP by Helsgaun (2000).

In this paper we aim to demonstrate that a well-implemented local search suffices to create a heuristic that can compete with the best heuristics in the literature. To this end we combine three powerful local search techniques, and implement them in an efficient way that minimizes computational effort. Furthermore, we make use of problem-specific knowledge, to guide the search to promising solutions more effectively. The heuristic created in this way not only performs well on many benchmark sets, it is also straightforward in its design and does not contain any components of which the contribution is unclear. The heuristic can also be readily applied to the VRP with multiple depots (MDVRP) and the VRP with multiple trips (MTVRP). The heuristic, together with benchmark instances, is available at `http://antor.uantwerpen.be/routingsolver/`.

In Section 2 we introduce local search components, and show how they can be implemented efficiently. In Section 3 we demonstrate how local search can be guided by the penalization of edges. Through various experiments we analyse different components of the resulting local search framework in Section 4, and extend it to problem variants in Section 5. In Section 6 we conduct detailed testing on the CVRP, the MDVRP and the MTVRP, and we conclude with a brief summary of our findings in Section 7.

2. Local search

Local search is one of few general approaches to combinatorial optimization problems with empirical success (Johnson et al., 1988). The basic idea underlying local search is that high-quality solutions of an optimization problem can be found by iteratively improving a solution using small (local) modifications, called *moves*. A *local search operator* specifies a move *type* and generates a *neighborhood* of the current solution. Given solution s, the neighborhood of a local search operator is the set of solutions $\mathcal{N}(s)$ that can be reached from s by applying a single move of that type.

After generating the neighborhood of the current solution, the neigborhood is evaluated, and local search uses a move *strategy* to select at most one solution from the neighborhood $\mathcal{N}(s)$ to become the next current solution. The most commonly used move strategy is called steepest descent, that selects the solution from the neighborhood with the best objective function value, provided that this solution is better than the current solution. If no improving solution is found in the neighborhood of the current solution, a *local optimum* has been reached.

In general, local search operators for the VRP can be distinguished between operators for *intra-route optimization* and operators for *inter-route optimization*. These two operator types reflect the two tasks that one has to solve in a VRP: (1) The allocation of customers to routes (inter-route optimization), and (2) the optimization of each route in itself (intra-route optimization). These two tasks do not necessarily have to be solved in this order, nor sequentially. For instance, an *Ejection Chain* (Glover, 1996) solves both tasks simultaneously. However, intra-route optimization can be executed rather efficiently, since it corresponds to solving a TSP with relatively few customers. Therefore, it seems sensible to optimize the routes in themselves, before they are optimised jointly.

The most commonly used operator for intra-route optimization is *2-opt*. The idea of 2-opt is to remove two edges from the considered route and replace them by two new edges, such that a new route is formed. The same idea can be extended to three edges (3-opt) and four edges (4-opt). However, the size of the considered neighborhood increases quickly in the number of considered edges. Lin and Kernighan (1973) (LK) have generalized this idea for the TSP in a computationally feasible way. LK has proven to be highly effective to solve TSPs, and therefore constitutes an ideal operator for intra-route optimization.

On the other hand, operators for inter-route optimization try to change the allocation of customers to routes. Let a route r_i be defined by the string (an ordered set) of customers that are visited on this route $r_i = \{I_1, I_2, \ldots, I_{|r_i|}\}$. The neighborhood of the operator *relocate* contains all solutions, in which a single customer is removed from its route and inserted into another route. More formally, it looks for a substring $\hat{r_i}$ of a route r_i, which is inserted into another route r_j, where the substring only contains a single element $|\hat{r_i}| = 1$. The operator *Or-exchange* generalizes this idea by removing and re-inserting longer substrings $|\hat{r_i}| \geq 1$. A *swap* tries to exchange two customers from two different routes. A substring $\hat{r_i}$ is exchanged with a substring $\hat{r_j}$ with $|\hat{r_i}| = |\hat{r_j}| = 1$.

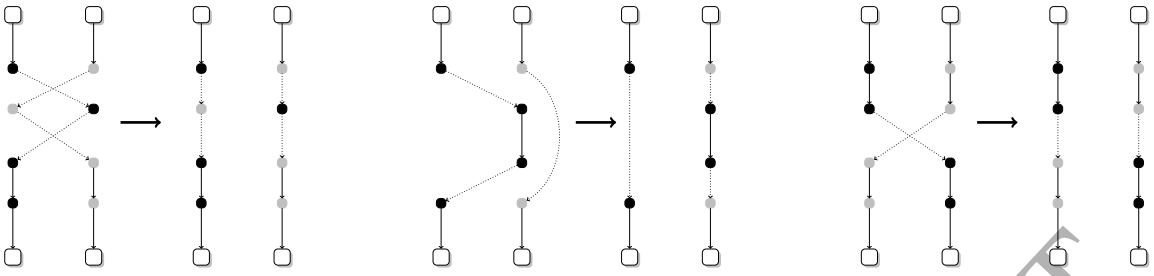


Figure 1: Illustrations of different local search moves. The routes are visualized vertically, the depot nodes are pairwise the same. From left to right: a swap (4 edges are exchanged), an Or-exchange (3 edges are exchanged), a Crossover (2 edges are exchanged).

This idea can again be generalized by exchanging larger substrings $|\hat{r_i}|, |\hat{r_j}| \geq 1$. In a *Crossover* both exchanged substrings are connected to the depot, and thus contain I_1 or $I_{|r_i|}$. Hereby, the substrings have possibly to be inverted. Finally, *CROSS-exchange* (CE) exchanges any two substrings of any size. Again, the substrings can be inverted (which is called I-CROSS, we denote both variants with CE in the following). Note that $|\hat{r_i}| = |r_i|$ and $|\hat{r_j}| = |r_j|$ would result in an exchange of two entire routes and the solution remains unchanged. These operators are the most commonly used and they are illustrated in Figure 1. Further operators include the ejection chain and the GENI insertion operator (Gendreau et al., 1992). Given this large pool of local search operators, there are some established principles as to which operators to choose in practice.

Complementarity

An important observation is that a local optimum for one local search operator is generally not a local optimum for another one. For this reason, it is sensible to use several local search operators in a single algorithm. This is the underlying principle of the *variable neighborhood search* metaheuristic framework (Mladenović and Hansen, 1997), but it is extremely common in the area of vehicle routing (Sörensen et al., 2008). For example, the active-guided evolution strategies of Mester and Bräysy (2007), use relocate, swap, 2-opt and Or-exchange, the hybrid genetic algorithm of Vidal et al. (2013b) uses relocate, swap and 2-opt, and the iterated local search of Subramanian et al. (2013) uses relocate, 2-opt, Or-exchange and CROSS-exchange. However, the usage of several local search operators is only beneficial as long as they explore different neighborhoods. In the best case, the pairwise intersections of the considered neighborhoods are empty, i.e., given two operators o_1 and o_2 we have $\mathcal{N}^{o_1}(s) \cap \mathcal{N}^{o_2}(s) = \emptyset$ for every solution s.

Complexity versus neighborhood size

In general, the computational complexity of a local search operator depends on the size of its neighborhood. The larger the neighborhood, the more solutions need to be generated and evaluated. On the other hand, larger neighborhoods also come with a larger probability of finding improvements. Consequently, there is a trade-off between computational complexity and the probability of improvement. This trade-off presents

one of the greatest challenges in the design of an efficient local search. A balance needs to be reached between the size of the neighborhood and the computational effort required to generate it. Operators with relatively small neighborhoods such as a swap can be easily implemented and quickly executed. However, they might not explore a sufficiently large neighborhood to find improvements. Reversely, operators with large neighborhoods such as CE exhibit a high complexity ($O(n^4)$). For larger instances already a quadratic complexity might be computationally too expensive. A commonly used tool to reduce the complexity of local search operators is *heuristic pruning*. Rather than generating the entire neighborhood for each operator, pruning tries to limit the considered neighborhood to promising options. As an example, CE usually only considers substrings up to a certain length, or a relocate-move can be restricted in such a way that only insertions next to relatively close nodes are considered. A metaheuristic that drastically restricts neighborhoods is, e.g., *granular tabu search* (Toth and Vigo, 2003). Another very effective approach to prune the neighborhood of more complex operators is sequential search (Irnich et al., 2006), which constitutes a generalization of the partial gains criterion in LK. The idea of sequential search is to split complex operators into exchanges of edges, and only consider subsequent exchanges as long as overall improvements are obtained. We provide a more elaborate description of sequential search below.

In summary, we argue that a successful local search for the vehicle routing problem contains a small yet diverse set of well-chosen local search operators for intra- and inter-route optimization that have sufficiently large neighborhoods, but are able to find improvements without exploring the neighborhood exhaustively. In the following, we outline how such an efficient implementation of local search operators can be realized, using the example of LK and CE. Both operators generate and evaluate rather large neigborhoods (LK incorporates 2-opt and 3-opt moves, CE includes insert, swap, and crossover). These operators can be complemented by an operator that can improve more than two routes simultaneously, and we introduce such an operator using the idea of embedded neighborhoods.

2.1. Lin-Kernighan Heuristic

Intra-route optimisation corresponds to solving a TSP in which the nodes are composed of the depot and all customers on the respective route. One of the most effective heuristics for the TSP is the heuristic by Lin and Kernighan (1973) (LK), which has been further refined by Helsgaun (2000) and Applegate et al. (2003) to solve instances with more than 100.000 customers. LK looks for κ-opt moves by exchanging κ existing edges with κ new edges. Since the complexity of finding κ-opt moves increases rapidly for larger κ, usually 2-opt and 3-opt are used in heuristics. LK introduces some clever ideas to also generate and evaluate more complex moves in a feasible runtime.

Starting with the longest edge in the considered route, edges are iteratively removed and added, such that the pair of removed and added edge shares an endpoint, fulfills the *partial gain criterion* and results in a feasible tour if the tour would be closed. With the partial gain criterion only those pairs of edges are considered for removal and insertion,

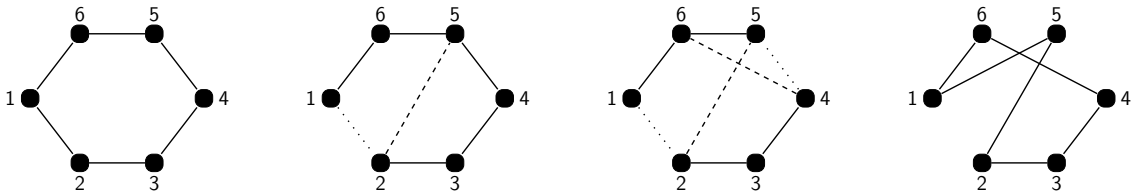


Figure 2: Illustration of the sequential removal and adding of edges in LK. Starting with the removal of one edge (1,2), an edge (2,5) starting from one of the endpoints is added, such that $c(1,2) > c(2,5)$. Because of the positive gain, the process continues by removing an edge incident to node 5, and adding an edge, e.g., (4,5) and (4,6). A continuation is considered, if $c(1,2) + c(4,5) > c(2,5) + c(4,6)$. The tour could be closed by removing edge (5,6) and adding edge (1,5) to obtain a 3-opt move.

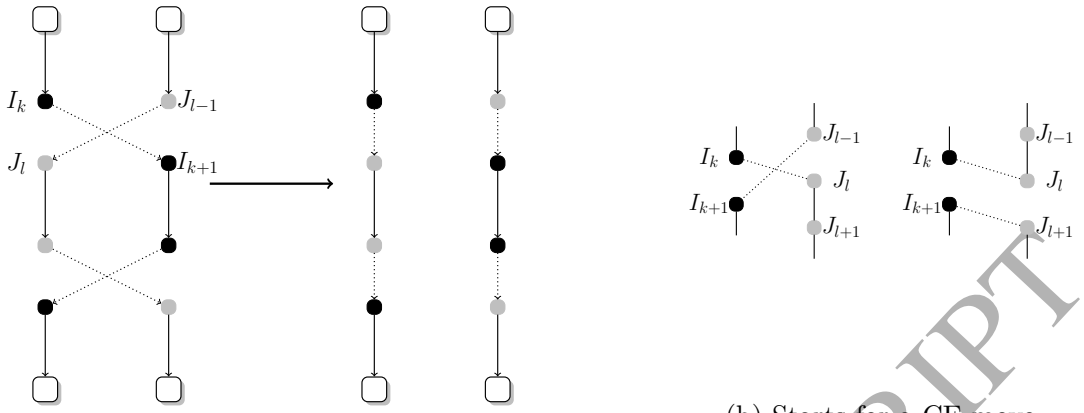
such that the overall sequence of exchanges results in an overall improvement. This criterion reduces the neighborhood drastically by restricting the search to promising options for larger κ. If, for instance, there is no sequential removal and insertion of two pairs of edges with a positive gain, LK does not continue to remove and add more pairs. An example is provided in Figure 2. As soon as an improving move is found with the closure of the tour, the move is executed and LK is restarted. The heuristic stops, if for each initially removed edge no improving move can be found. For a more elaborate description of LK we refer the interested reader to Helsgaun (2000).

As in previous LK implementations, we prune the neighborhood by only considering new edges between a node and its ten nearest nodes within the considered route. Routes in VRPs tend to be rather small TSPs with few customers so that we slightly adapt the standard version. Instead of executing the first improving move, we generate and evaluate all feasible κ-opt moves (starting with the removal of one particular edge) and execute the best improving move. The upper limit for κ is set to four.

2.2. CROSS-exchange

The CROSS-exchange operator (CE) is a generic local search operator that tries to exchange two substrings $\hat{r_i}$ and $\hat{r_j}$ of two different routes r_i and r_j (Taillard et al., 1997). An example of such a move is visualized in Figure 3. The generation of all substrings of a route has the theoretical complexity $O(n^2)$, and thus matching two substrings amounts to $O(n^4)$. To tame this complexity, the length of considered substrings is usually limited. In the following we adopt the idea of partial sums from LK and sequential search to efficiently prune the neighborhood of CE.

The evaluation of a CE move can be deconstructed into two parts: (1) The identification of the start of two substrings and (2) the determination of a suitable length of both substrings. To find the start of a substring that is potentially removed from route r_i, one first need to determine an edge that is removed. Let this edge be (I_k, I_{k+1}). For I_k a new neighbour in another route needs to be found. Using the idea of heuristic pruning we only consider potential neighbours among the C closest nodes. Let J_l be such an option, a node in a different route that is among the C closest of I_k. If (I_k, J_l) is added as a new edge, one of the edges (J_l, J_{l-1}) or (J_l, J_{l+1}) needs to be removed, since J_l can only have two adjacent nodes. Finally, the incident node of the removed



(a) A complete CE move.

(b) Starts for a CE move.

Figure 3: (a) Illustration of a CE move with substrings of two customers. (b) Two possibilities to generate starts for a CE. Starting with the removal of (I_k, I_{k+1}), we add an edge (I_k, J_l). We can either complete the start by removing (J_l, J_{l-1}) and adding (I_{k+1}, J_{l-1}), or by removing (J_l, J_{l+1}) and adding (I_{k+1}, J_{l+1}). Starts are considered, if the sum of the costs of added edges is not higher than the sum of costs of removed edges. In both cases, the substrings can start from I_{k+1} and J_l (see left), or from I_k and J_{l-1}.

edge J_{l+1} or J_{l-1} is reconnected to I_{k+1}. A visualization is given in Figure 3. In total, at most $4C$ starts are evaluated for a CE move which starts with the removal of one edge (I_k, I_{k+1}) ($2C$ for all nearest nodes of I_k and $2C$ for all nearest nodes of I_{k+1}).

For each start generated this way, we only continue with (2) the determination of substrings, if the start does not increase solution costs, e.g., $c_1 = c(I_k, J_l) + c(I_{k+1}, J_{l-1}) - c(I_k, I_{k+1}) - c(J_l, J_{l-1}) \leq 0$. The start generates a first 'cross' between two routes, and can already be stored as a candidate move if route constraints are not violated (a crossover). For a CE move we need to determine a second 'cross' between the two routes. Starting form the first 'cross', substrings can be constructed in two ways. The start of the substring to be inserted into r_j can either be I_k or I_{k+1}. Let $\hat{r_i} = \{I_{k+1}\}$ be the substring to be inserted into r_j and $\hat{r_j} = \{J_l\}$ be the corresponding substring to be inserted into r_i (the description for $\hat{r_i} = \{I_k\}$ and $\hat{r_j} = \{J_{l-1}\}$ follows analogously). Then the cost of the second 'cross' is $c_2 = c(I_{k+1}, J_{l+1}) + c(J_l, I_{k+2}) - c(I_{k+1}, I_{k+2}) - c(J_l, J_{l+1})$. If $c_1 + c_2 \leq 0$, a candidate move has been found (a swap). We repeat this evaluation for different lengths of the substrings. Keeping $\hat{r_i} = \{I_{k+1}\}$ fixed, we extend $\hat{r_j} = \hat{r_j} \cup J_{l+1}$, recompute c_2 with $l = l + 1$, and add it as candidate move if $c_1 + c_2 \leq 0$. This extension of $\hat{r_j}$ is continued until the depot node is reached, or until the capacity constraint of r_i is violated. We then iteratively extend $\hat{r_i}$, starting with $\hat{r_i} \cup I_{k+2}$, and repeat the above process. In this way, we determine all combinations of substrings starting from I_{k+1} and J_l, whose exchange result in a non-increasing solution value while meeting route constraints.

The special cases $\hat{r_i} = \emptyset$ or $\hat{r_j} = \emptyset$ (an Or-exchange) can be generated and evaluated analogously with different functions c_1^* and c_2^*. Using the example above with $\hat{r_i} = \emptyset$, we have $c_1^* = c(I_k, J_l) - c(I_k, I_{k+1})$. If $c_1^* \leq 0$, we again evaluate all extensions of $\hat{r_j}$

until the depot node is reached or constraints in r_i are violated. For $\hat{r_j} = \{J_l\}$ we have $c_2^* = c(I_{k+1}, J_l) + c(J_{l-1}, J_{l+1}) - c(J_l, J_{l-1}) - c(J_l, J_{l+1})$.

In summary, we suggest the following approach to iteratively generate and evaluate the neighborhood of the CE operator, starting with the removal of edge (I_k, I_{k+1}) :

Algorithm 1 CROSS-exchange with sequential search.

1: Determine the nearest nodes of I_k and I_{k+1} that are in a different route r_j.
2: Determine all starts with $c_1 \leq 0$ and $c_1^* \leq 0$.
3: For each remaining start, iteratively evaluate and extend the substrings. If the exchange results in feasible routes and $c_1 + c_2 \leq 0$, store the move as a candidate move (CE), analogously for $c_1^* + c_2^* \leq 0$ (Or-exchange).
4: Execute the candidate move with the steepest descent.

This approach generates a potentially very large neighborhood, especially since we do not impose a restriction on the length of substrings. However, the evaluation of starts runs in linear time, and is likely to remove many options, especially in good VRP solutions.

2.3. Complementary compound moves - Relocation chain

CE generates large neighborhoods to improve a pair of routes simultaneously, while LK optimises the individual routes. These two operators can be complemented by an operator that affects more than two routes simultaneously. Such an operator can be constructed by using the ideas of an *embedded neighborhood*. In an embedded neighborhood several simple moves are combined to form one compound move (Ergun et al., 2006). An example for a compound move is the ejection chain introduced in Glover (1996) and formalized in Rego (2001) for the VRP.

An ejection chain can induce changes in several routes simultaneously, however, it also generates intra-route moves. With LK we have a dedicated operator for intra-route optimisation, and thus a complementary operator should focus exclusively on changes between routes. We build such an operator by combining several relocation moves. This idea is similar to the concept of an ejection chain, however, we only allow inter-route relocations. A relocation is simple to implement and to evaluate, and thus, it constitutes a suitable building block for a more complex compound move. In the following we will call this compound move *relocation chain* (RC).

A RC starts with a relocation of a customer node from route r_i into route r_j. This relocation is followed by a relocation of a customer node from route r_j into route r_k (where $i = k$ is possible). This process can be repeated until an upper limit of relocations is reached. The motivation for such an operator is to change the solution while maintaining feasibility. Especially in tightly-constrained VRPs, many pairwise exchanges between routes will violate constraints, so that the neighborhood of feasible solutions can be relatively small. A relocation of a node from r_i into r_j might improve the solution, but exceed the capacity constraints of r_j. However, the move might

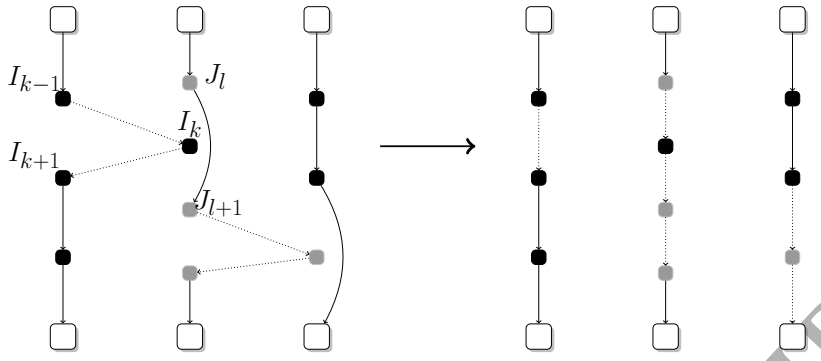


Figure 4: Illustration of a relocation chain with two relocations.

become feasible, if we make space by simultaneously relocating a node from r_j into another route. An example is visualized in Fig. 4.

The complexity of this compound move is relatively high. Let us assume that we start a RC with node I_k. We can relocate this node into any position of any other route ($O(n)$). Let r_j be an option for a destination route (there might be several options). Then we can continue the RC by relocating any node in r_j into any other position of any other route ($O(n^2)$). Thus, with every continuing relocation, the complexity of the operator grows exponentially.

This complexity can be reduced with ideas from sequential search, pre-processing and heuristic pruning. Let a RC start with node I_k. Then the costs of a relocation of I_k next to J_l can be computed as the difference between the minimal costs of inserting I_k either in front or behind of J_l, and the detour induced by the visit of I_k in its current route. The insertion cost can be computed as $c_I = \min_{J_l^* \in \{J_{l-1}, J_{l+1}\}} \{c(I_k, J_l) + c(I_k, J_l^*) - c(J_l, J_l^*)\}$ and the cost of the detour as $c_D = c(I_{k-1}, I_{k+1}) - c(I_k, I_{k+1}) - c(I_k, I_{k-1})$.

This cost information can be preprocessed at the start of the heuristic and updated when necessary. If the relocation does not increase solution costs and we have $c_I + c_D \leq 0$, we consider it as start of a RC. This corresponds to the idea of partial gains in LK and the basic idea of sequential search. If the relocation keeps r_j feasible, we store it as candidate move. Otherwise, we try to extend the RC by a relocation of a node in r_j. A subsequent relocation is considered, if it restores feasibility in r_j, and the overall costs of both relocations are non-positive. If the destination route of the second relocation is feasible, we store the set of both relocations as a candidate move. Otherwise, we continue the same process until an upper bound of relocations is reached.

In other words, we sequentially relocate nodes in such a way that the aggregated costs are non-positive and all but the destination route of the last relocation are feasible. For every relocation there might be several options for a continuation move, and thus, information about moves and their continuations is stored in a tree-structure. In order to avoid further cost computations, we do not allow an insertion into the position of a previously ejected customer node in the same RC. As in CE, we further reduce complexity by only considering insertions of a node next to its C closest customer nodes. The more of those nodes are in the same route, the less options remain. Additionally,

we only consider one relocation position per destination route. If, for instance, two of the nearest nodes of I_2 are J_3 and J_5, we only consider the relocation of I_2 into the position of route r_j that results in the lowest insertion cost.

After, for each starting relocation, we have explored an upper limit of following relocations, we execute one of the candidate moves. A simple idea to improve efficiency and make use of the possibly quite long list of generated candidate moves, is to execute multiple moves. We first execute the candidate move with the largest decrease in costs (steepest descent), and then remove all other candidate moves that 'interfere' with the executed move. A move interferes with another move, if it includes a relocation of a node that is also relocated in the other move, or a relocation of a node that is a new or old neighbor of a node relocated in the other move. This interference changes the cost evaluation of the respective moves and would require additional cost computations. Among all non-interfering moves we again execute the one with the largest decrease in costs, and so forth.

We observed during the experiments described in Section 4 that the computation of RCs with a maximal length of four relocations requires up to 10 times more computation time than RCs with maximal three relocations. To keep computations efficient, we therefore limit RCs to three sequential relocations.

3. Guiding local search

The three operators LK, CE and RC generate large and diverse neighborhoods. Yet, at some point they will inevitably reach a local optimum s^* for which the generated neighborhoods $\mathcal{N}(s^*)$ do not contain a better solution. Heuristics usually use *perturbation* to escape this local optimum. The idea of perturbation is to change solution s^* where, in contrast to local search, this change does not necessarily have to result in an improving solution. Perturbation can appear in many different forms. *Large neighborhood search* and *ruin & recreate* destroy and repair parts of the solution. Methods based on simulated annealing accept, with a certain probability, local search moves that worsen the solution. Generally, most heuristics utilize some form of randomization to achieve a perturbed solution, or a pool thereof.

One approach that does not rely on randomization nor on additional algorithmic components is *guided local search* (GLS) (Voudouris and Tsang, 2003). Even though GLS is not as popular as other metaheuristics (for instance tabu search or variable neighborhood search), it has been successfully applied to the TSP (Voudouris and Tsang, 2003) and the VRP (Mester and Bräysy, 2007).

The idea behind GLS is to change the cost evaluation of s^*, rather than s^* itself. Features that are considered bad are penalized. In the context of the VRP, those features are edges, and the costs of bad edges are increased. More formally, we change the cost $c(i, j)$ of an edge in the current solution between customers i and j to

$$c^g(i, j) = c(i, j) + \lambda p(i, j) L, \tag{1}$$

where $p(i,j)$ counts the number of penalties of edge (i,j), L is a proxy for the average cost of an edge, computed as the costs of the starting solution divided by the number of customers, and λ controls the impact of penalties. Kilby et al. (1999) experimentally found $\lambda \in [0.1, 0.3]$ to be a good choice while Mester and Bräysy (2007) use $\lambda = 0.01$. We confirmed in pre-tests that $\lambda = 0.1$ works well on the tested instances in Section 4, and use this value in the remainder of the paper.

After the penalization of an edge, local search can be used on the adapted search space to potentially remove this edge. With increasing costs, the probability of finding an c^g-improving move increases. Note that a c^g-improving move does not have to correspond to a c-improving move.

The effectiveness of this penalization strategy hinges on the identification of 'bad' edges. An obvious idea is to focus on the most expensive edges as in Mester and Bräysy (2007), since these contribute more weight to the objective function. This simple observation seems to constitute the entire body of knowledge on which features of a VRP solution should be considered 'bad', or, more precisely, which edges should be removed preferentially.

In order to shed some light on this issue, we have performed an exploratory study that attempted to find common characteristics of both good and bad solutions, in order to decipher which properties are likely to appear in good solutions and not in bad ones or vice versa. Details of both the methodology and the results of this study are beyond the scope of this paper, and can be found in Arnold and Sörensen (2018). First, we generated sets of instances with varying instance attributes, such as the number of customers and the positioning of the depot. For each instance we computed a near-optimal solution, as well as a non-optimal solution. Both solutions were computed with a GLS similar to the one used in Section 5 and a classical edge penalization. The non-optimal solutions were generated to have a pre-defined gap to the near-optimal value of either 2% or 4%, while the near-optimal solutions were expected to have a small gap to the optimal solution (on a benchmark set of similar instances the average gap was computed as 0.20%). We then defined several metrics to transform the structure of a solution into quantitative metrics. These metrics were largely based on geometric properties, such as the width and depth of routes (see further for definitions), or the number of intersecting edges. In total, we generated and quantified about 192.000 solutions for 96.000 different VRP instances. We then used a classification learner to discover metrics that distinguish near-optimal from non-optimal solutions.

The most predictive metrics, as to whether a solution is near-optimal or non-optimal, were the width and compactness of routes. Solutions of higher quality appear to have narrower and more compact routes. Hereby, the *width* of a route is defined as the maximum distance between any pair of customers, measured along the axis perpendicular to the line connecting the depot and the center of gravity of a route. The coordinates of the center of gravity of a route are calculated as the average of the coordinates of all nodes in that route. The *compactness* of a route is the average distance to the line connecting the depot and the center of gravity of all customers in that route. Other metrics with a lower predictive power included the number of intersections (edges in

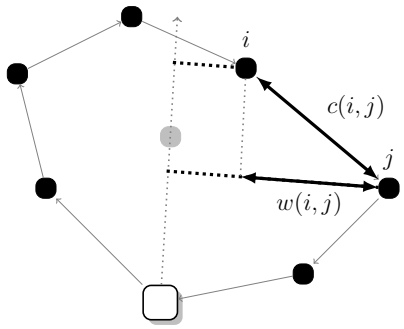


Figure 5: Metrics determining the 'badness' of edge (i, j). The *width* $w(i, j)$ is computed as the distance between nodes i and j measured along the axis perpendicular to the line connecting the depot and the route's center of gravity (gray dot). The *cost* $c(i, j)$ is equal to the length of the edge.

different routes that cross each other) and the average length of edges connected to the depot.

In other words, moves that reduce the width of a route or increase its compactness are likely to make the solution better. Within the context of a GLS, we need to translate these general findings on bad solutions into guidelines on bad edges. Firstly, wider and less compact routes tend to have wider edges, and thus, it seems straight-forward to penalize and thereby avoid *wide* edges. We adopt the same definition for the width of an edge as in Arnold and Sörensen (2018), and compute the width of an edge as the distance measured along the axis perpendicular to the line connecting the depot with the center of gravity of the route. Secondly, the penalization of long edges, i.e., edges with a high *cost*, seems reasonable since it has already proven to work well in the context of the VRP and the TSP. We confirmed with the above study that especially those edges that are connected to a depot tend to be shorter in high-quality solutions.

These two metrics *width* and *cost* are illustrated in Figure 5, and they can be readily transformed into functions that measures the 'badness' of an edge (i, j):

$$b^w(i,j) = \frac{w(i,j)}{1+p(i,j)} \qquad b^c(i,j) = \frac{c(i,j)}{1+p(i,j)} \qquad b^{w,c}(i,j) = \frac{w(i,j)+c(i,j)}{1+p(i,j)} \tag{2}$$

where $w(i, j)$ denotes the width of edge (i, j), and $c(i, j)$ its cost. The division by the number of previously received penalties is a standard approach in GLS to enhance the diversification of the penalization, and to avoid that the same edge is penalized over and over again.

A penalization strategy combined with a set of implemented local search operators (LS) are sufficient to build a GLS heuristic for the VRP, as outlined in Algorithm 2. After the construction of a starting solution, LS is applied on the solution until a local optimum is reached. The local optimum is perturbed by iteratively penalizing and attempting to remove edges. After an adequate number of changes have been made in this phase, the resulting solution is re-optimised with LS. Hereby, 'apply LS on a certain search space' means that only moves are considered that originate from this space, e.g.,

if the search space is a single edge, only CE moves that start with the removal of this edge are considered, or RC moves that start with a relocation of one of the incident nodes. This heuristic framework is entirely based on local search, and does not depend on other components. Perturbation is achieved by changing the evaluation criterion and thereby inducing changes in the local optimum. The more moves are executed during this phase, the more the solution is perturbed. Note that a move is only executed if it improves the solution under $c^g(\cdot)$, and thus, not every penalization has to result in a subsequent move. Finally, the perturbed solution is optimised under normal evaluation $c(\cdot)$. We only apply LS on the part of the search space that changed during the previous perturbation phase, since routes that were not affected are unlikely to be the starting point of improving moves. This framework poses some interesting research questions which we will investigate in the following. Which operators should LS contain, how much perturbation is necessary and which edges should be penalized?

Algorithm 2 A simple GLS framework.

1: *Construction.* Construct a starting solution.
2: *Initial optimisation.* Apply LS on starting solution.
3: **while** not abortion criterion is reached **do**
4: *Perturbation*:
5: **while** not a certain number of moves have been made **do**
6: Penalize 'the worst' edge (i, j) by incrementing $p(i, j)$.
7: Apply LS on (i, j), using $c^g(\cdot)$ as evaluation criterion.
8: **end while**
9: *Optimisation.* Apply LS on all routes that were changed during perturbation (using $c(\cdot)$ as evaluation criterion).
10: **end while**

4. What makes a local search effective?

In the following we conduct a series of experiments to explore various aspects of local search within the above GLS. We compare different neighbourhoods, different magnitudes of perturbation as well as different strategies to penalize edges. Three different sets of complementary local search operators are used as shown in Table 1. Set LS_1 is composed of relatively simple operators and generates the smallest neighboorhods of all sets. We mirror these operators with our CE implementation, where only exchanges of substrings of length one are considered (swap) or exchanges in which one substring is empty (relocate and Or-exchange). For the intra-route optimisation we use our LK implementation, but only consider 2-opt moves. LS_2 generates larger neighborhoods for both, intra-route and inter-route optimisation and uses the operators as described in the previous section. In LS_3 we additionally use the relocation chain as complementary neighborhood, so that this set should generate the largest neighborhoods.

Table 1: Local search operators used in the different setups.

	Intra-route LS	Inter-route LS
LS_1	2-opt	relocate, swap, Or-exchange
LS_2	LK	CE
LS_3	LK	CE, RC

Each of the three LS variants is used within the GLS framework. A starting solution is constructed with the popular and relatively simple heuristic by Clarke and Wright (1964). During the optimisation phase, we first apply the inter-route operators on the considered search space, execute the move with the most beneficial evaluation (steepest descent), and then use the intra-route operator to optimise the changed routes. We iteratively apply this combination of inter- and intra-route optimisation until in one iteration the solution is not improved.

During the perturbation phase we iteratively penalize an edge and apply the inter-route operators on this edge, i.e., only moves which start with the removal of the respective edge are considered. Since the perturbation should trigger changes in the allocation of customers to routes, we apply the intra-route operator only at the end of the phase.

All experiments are executed on instances 26 to 55 by Uchoa et al. (2017). These instances have a reasonable size with 219 to 367 customers and are diverse with respect to all important characteristics of a VRP instance: clustering of customers (strong clustering versus uniform distribution), variation in customer demand ($[1, 1]$ to $[1, 100]$), the average length of routes (from 3 customers per route up to 24 customers per route) and depot location (central location versus eccentric location). Thus, experiments on this diverse set of instances should reduce the danger of overfitting the heuristic to particular types of instances. We report the results as performance over time on all 30 instances. The performance is expressed as the average gap between the best solutions found within a specific time horizon and the best known results, as reported by Uchoa et al. (2017). The average gap of the starting solutions computed with Clarke and Wright is about 6.4%.

4.1. Neighborhood Size and Perturbation

We investigate the impact of different neighborhoods generated by LS_1, LS_2 and LS_3 in interplay with different magnitudes of perturbation, where $P \in \{10, 100, 1000\}$ represent the number of moves that are executed in the perturbation phase. The edge penalization strategy b^c, and the degree of pruning for inter-route operators $C = 30$ are kept fixed. From the results in Figure 6 we can make the following observations.

Firstly, larger neighborhoods find better solutions. This seems a straight-forward statement to make, however, note that usually larger neighborhoods come at the expense of higher computational effort. Thus, smaller neighborhoods should generally find improving moves faster, at least at the start of the search. Since we do not observe

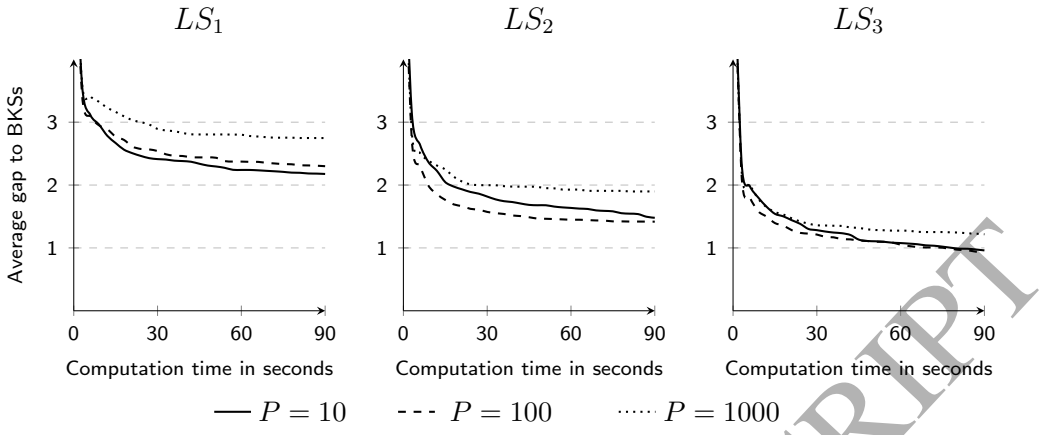


Figure 6: Average gap (in %) to the best known solutions (BKSs) over time on 30 instances by Uchoa et al. (2017) for different setups.

this effect, we can conclude that the larger neighborhoods are pruned effectively and exhibit a good time-quality trade-off. On the other side, the results show that even a minimal local search implementation of four simple operators can obtain solutions within a 2% range in a very short computation time. Thus, gaps of this size should constitute a minimal standard when evaluating the performance of a new heuristic.

Secondly, RC seems to successfully complement CE, since the addition of this operator results in significant performance improvements. This is especially remarkable given that RC requires the most computation time of all three operators. Depending on the average length of routes, RC requires between 40% (on instances with longer routes) up to 80% (on instances with shorter routes) of the entire computation time. This additional computational effort seems to be beneficial.

Finally, the degree of perturbation should be chosen within reasonable bounds. Even though little perturbation ($P = 10$) seems to work well for smaller neighborhoods, for larger neighborhoods slightly more perturbation appears to be a better choice. An overly large perturbation ($P = 1000$) appears to disintegrate the previous local optimum for all considered neighborhoods, and results in a worse performance. Interestingly, the differences between different parameter choices become smaller with larger neighborhoods, as if larger neighborhoods compensate for a 'poorer choice'. We conclude that good results can be obtained with rather little but not too little perturbation. In more granular tests we found the best results with $P = 30$ for LS_3, and we will use this value in the remainder of the paper.

4.2. Penalization criterion

Keeping the best local search setup LS_3 with $P = 30$ fixed, we investigate which edges should be penalized. We compare five different strategies. The first three strategies correspond to b^w, b^c, and $b^{w,c}$. The fourth strategy diversifies the penalization, and deterministically changes the penalization criterion after every perturbation phase, in this order (rotation). Finally, we investigate the impact of the guided penalization

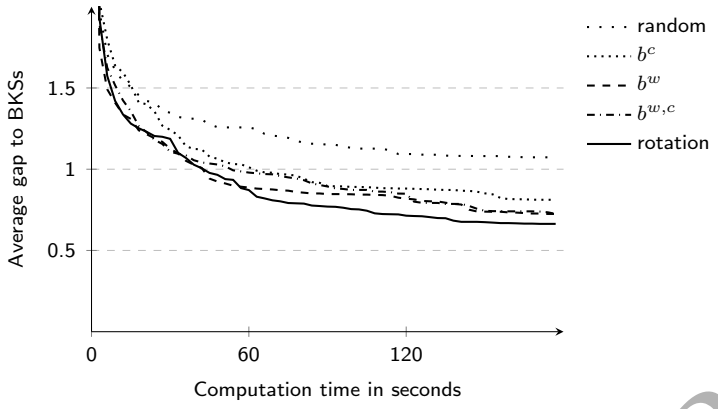


Figure 7: Average gap (in %) to the best known solutions (BKSs) over time on 30 instances by Uchoa et al. (2017) for different setups.

strategies above by comparing them to an unguided strategy, in which the penalized edge is chosen randomly (the seed of the random generator is fixed to keep the deterministic behaviour of the heuristic).

The results in Figure 7 reveal that the choice of edge penalization has less impact on performance than the choice of local search operators. Even with a randomized penalization solutions with an average gap of about 1% can be computed. However, guidance can still improve this already good performance of the heuristic. Hereby, width appears to be a slightly better criterion to detect bad edges than cost, which confirms our previous findings. The combination of both features $b^{w,c}$ yields similar results, while the best performance is obtained with a diversified penalization. A possible explanation for the success of the rotation strategy is that the best choice might be instance-dependent. For some instances it might be better to remove long edges, while for other ones it might be more important to obtain tight routes. Another explanation could be that the exclusive focus on one criterion narrows the search, and some edges are targeted excessively while others remain untouched. In other words, the degree of diversification might matter. We investigated the impact of diversification by testing the rotation strategies $b^w - b^c$ (less diversification) and $b^w - b^c - random$ (more diversification). In both cases, we obtained worse results than with $b^w - b^c - b^{w,c}$, and thus, this rotation strategy appears to be a sweet spot for the tested instances.

In conclusion, the penalization strategy has less impact on performance than the local search, which suggests that most of the heuristic's efficiency is driven by a well-implemented local search. Nevertheless, guidance with problem-knowledge still elevates the overall performance.

4.3. Pruning

For the inter-route operators we only consider new edges between a node and its C closest nodes. For the experiments above we chose $C = 30$, since this value appeared to yield good result in pre-tests. In the following we compare different pruning setups. If

the value for C is lower, then the pruning is tighter and the local search operators should be faster so that the GLS can execute more iterations in the same amount of time. On the other hand, a looser pruning increases the size of the considered neighborhoods at the expense of higher computational effort.

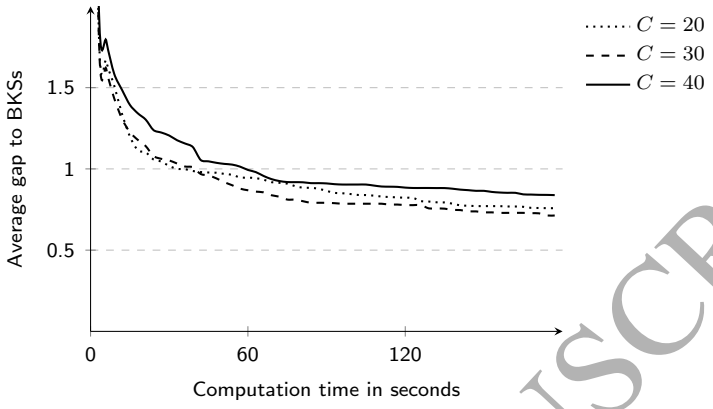


Figure 8: Average gap (in %) to the best known solutions (BKSs) over time on 30 instances by Uchoa et al. (2017) for different setups.

From the results in Figure 8 we observe that a tighter pruning is effective within short computation times, while $C = 30$ seems to be the better choice for longer runtimes. $C = 40$ appears to be too loose on the tested instances, and the size of the explored neighborhoods becomes unnecessarily large. We also observed some minor differences with respect to instance features. Generally, a looser pruning appeared to work better for instances with relatively small routes or instances with a large variation in demand. However, these effects were rather weak, and for simplicity we choose the same degree of pruning for all instances in the following.

Instead of considering the C nearest nodes in terms of cost $c(\cdot)$, one could also use a different metric to define nearness. A good example is the α-nearness defined in Helsgaun (2000) for the TSP. The α-nearness $c^\alpha(\cdot)$ between two nodes i and j is defined as the cost difference between the optimal 1-tree of all nodes and the minimum 1-tree that contains edge (i, j). Since every TSP solution is a 1-tree, both problems have a certain degree of similarity, but a minimal 1-tree can be computed more efficiently than a TSP solution. We implemented this idea for the VRP, and construct the 1-trees out of all customer nodes. We found that on average 96% of the $C = 30$ nearest nodes in terms of $c(\cdot)$ were also nearest nodes in terms of $c^\alpha(\cdot)$. As a consequence, the local search considers the same edges and obtains the same performance. We also tested another metric based on the idea of width. Given two nodes i and j, we compute the distance between i and the line connecting the depot and j (or reversely if the corresponding distance is shorter), and add this 'width' to the cost $c(\cdot)$. The intuition behind this metric is to connect customers that are on the same line outgoing from the depot, and thus to obtain little detours. We found that this metric yielded good performances, however, it did not perform better than the simple $c(\cdot)$ metric. In conclusion, the

traditional pruning with respect to costs appears to be an efficient pruning strategy for the VRP.

4.4. Inital route minimization

For some VRP instances it is crucial to minimize the number of vehicles used. Even if the minimisation of routes is not part of the objective function, usually good solutions have as few routes as possible, since each additional route requires an additional edge and is thereby likely to increase the overall distance traveled. None of the above local search operators explicitly attempts to pursue this task and remove routes, even though a route might be eliminated accidentally. As a consequence, it appears sensible to minimize the number of used vehicles already in the construction phase.

We realize the minimisation of routes in the following way. We first compute the lower bound of vehicles $M_{min} = \lceil \frac{D}{Q} \rceil$ (where D is the sum of all demand and Q the capacity limit), and then construct a solution with the parallel version of CW that requires M_{CW} routes. The parallel version of CW generally constructs good solutions but might not minimize the number of routes. If we have $M_{CW} > M_{min} + 1$, we conclude that this instance is rather difficult with respect to the bin-packing problem of assigning customers to routes. In this case, we construct a second solution with CW with one minor modification. Some successful heuristics for bin-packing algorithms like *First Fit Decreasing* (Simchi-Levi, 1994) try to group large items first, and we translate this intuition to the VRP by trying to assign customers with a high demand first. We compute the weighted saving $s_w(i,j)$ of edge (i,j) as $s_w(i,j) = \frac{s(i,j)}{max_{k,l} s(k,l)} + \frac{d(i)+d(j)}{max_{k,l} d(k)+d(l)}$, where $s(i,j)$ denotes the saving as computed in the classical CW heuristic, and $d(i)$ denotes the demand of customer i. All weighted savings are then sorted in descending order, and used as input of the parallel CW. Thus, edges between customers with a high demand are prioritised. This approach successfully reduced the number of routes in the starting solution of 33 out of 100 instances in the benchmark set by Uchoa et al. (2017), and in some cases improved the solution quality significantly.

4.5. The complete local search heuristic

On the basis of the experimental results we identified the following local search setup as the most effective one, which we denote knowledge-guided local search (KGLS).

We assume that larger instances with more customers require more computation time and, thus, we define the abortion criterion as a maximum runtime with respect to the number of customers. The heuristic is entirely based on a local search with the steepest descent acceptance criterion, and thus, it works in a completely deterministic fashion. This determinism simplifies its analysis and the evaluation of its performance. In contrast, most state-of-the-art heuristics utilize randomness to diversify the search and escape local minima (Gutjahr, 2010). The exact role and the benefits of including random elements are rarely investigated, notwithstanding the fact that reliance on randomness has some significant disadvantages and, according to some researchers, can prevent the development of better, deterministic search components, see e.g., Glover (2007).

Algorithm 3 Knowlege-guided local search heuristic (KGLS).

Construction. Construct a starting solution with the CW heuristic. If $M_{CW} > M_{min} + 1$, re-compute the starting solution with adapted savings $s_w(i, j)$. Optimise the individual routes with LK.
Initial optimisation. Apply CE and RC on starting solution. Whenever a route is changed, re-optimise it with LK. Set $b(\cdot) = b^w(\cdot)$.
while time limit not reached **do**
    Perturbation:
    while not $P = 30$ moves have been made **do**
        Penalize edge (i, j) with the highest value $b(i, j)$ by incrementing $p(i, j)$.
        Apply CE and RC on (i, j), using $c^g(\cdot)$ as evaluation criterion.
    end while
    Optimisation. Apply LK, and then iteratively CE and RC on all routes that were changed during perturbation ($c(\cdot)$ as evaluation criterion). Whenever a route is changed with CE or RC, re-optimise it with LK. Change $b(\cdot)$.
end while

5. Extension to routing variants

In the following, we demonstrate with two examples how the efficiency and relatively straight-forward design of the knowlege-guided local search heuristic can be used to tackle other routing variants.

5.1. Multi-Depot Vehicle Routing Problem

In the Multi-Depot Vehicle Routing Problem (MDVRP) customers can be delivered from a given set of depots (Montoya-Torres et al., 2015). This adds another decision level: not only does the heuristic have to assign and sequence customers in routes, it also needs to determine which routes start and end at which depot. Notwithstanding this additional decision level, we can readily apply the heuristic to the MDVRP. The initial solution is constructed with a greedy approach, where each customer is assigned to its closest depot, and the routes per depot are then computed with the Clarke and Wright heuristic. The MDVRP also involves to determine a suitable number of routes for each depot, since it is not clear how many and which routes should originate from which depot. KGLS can remove routes, but it does not create new ones. This is sufficient for VRP instances in which one usually aims at minimizing the number of routes. For MDVRP instances, however, it is sensible to also allow the creation of new routes for a depot, and thereby enable changes in the assignment from routes to depots. We add one empty dummy route per depot, and allow RC to relocate customers into these empty routes, as long as the respective move improves the solution.

5.2. Multi-Trip Vehicle Routing Problem

An implicit assumption of the VRP is that each vehicle is responsible for one route. In some practical applications several tours might be assigned to the same vehicle,

e.g., in the context of city logistics smaller vehicles are often used to deliver customers on several short routes. Such scenarios can be modelled with the Multi-Trip Vehicle Routing Problem (MTVRP) which extends the VRP by adding the following constraint: Routes have to be assigned to M vehicles in such a way that the total cost of the routes assigned to the same vehicle does not exceed a time horizon T (denoted *time horizon constraint* in the following). T could for instance define the duration of a typical working day. For a more elaborate description of the problem we refer to Cattaruzza et al. (2016). Given an MTVRP, a solution for the corresponding VRP (without the time horizon constraint) is also a feasible solution for the MTVRP, if a feasible assignment of routes to vehicles can be found. Finding such an assignment represents a bin-packing problem which can usually be solved with simple heuristics in a short time.

KGLS performs many slight changes without violating route constraints, and thus, it generates a large number of routing solutions in a short time. Therefore, a simple idea to solve an MTVRP is to transform it into the corresponding VRP, solve the VRP with KGLS, and check for each computed solution whether it satisfies the time horizon constraint. In short, we generate a large pool of different VRP solutions and hope to find at least one that satisfies the additional constraints, an approach often used in the literature (Cattaruzza et al., 2016). The only modification, which is necessary to incorporate this approach in the knowledge-guided local search heuristic, is the implementation of a bin-packing heuristic. More concretely, each solution obtained after the perturbation and the optimisation phase in Algorithm 3 is evaluated with the three simple bin-packing heuristics *First Fit*, *First Fit Decreasing* and *Best Fit* (Simchi-Levi, 1994). If any of these heuristics find a feasible bin-packing solution that satisfies the time horizon constraint, then the respective VRP solution is a candidate solution for the MTVRP. The best candidate solution computed after maximal runtime constitutes the solution for the MTVRP.

6. Computational Results

We test the performance of KGLS on the entire instance set by Uchoa et al. (2017) ($\mathbb{U}$). As explained above, this instance set covers a wide problem variety and considers different degrees of clustering customers, variation in customer demand, and different depot locations. For the MDVRP we perform the experiments on the benchmark set by Cordeau ($\mathbb{C}$) (Cordeau et al., 1997). Examples of solutions for both instance sets are visualized in Figure 9. Finally, we use the benchmark set by Taillard et al. (1996) to investigate the performance on MTVRP instances.

We compare the performance of KGLS with some of the most efficient VRP heuristics in the literature: the hybrid genetic algorithm (HGSADC) of Vidal et al. (2013b), the iterated local search (ILS) of Subramanian et al. (2013), and the classical adaptive large neighborhood search (ALNS) of Pisinger and Ropke (2007). For the MTVRP we use the results from the ALNSP by François et al. (2016) and the memetic algorithm (MA) by Cattaruzza et al. (2014) as comparison. Whereas KGLS produces the same solution in every run, all other heuristics include random components, and we

present their results as the average performance over a number of runs[2]. The results of HGSADC and ILS are reported as the average performance over 50 runs, the results of ALNS display the average over 10 runs, and the results of ALNSP and MA present the average performance over 5 runs.

Figure 9: Benchmark instances from the $\mathbb{U}$-set and the $\mathbb{C}$-set.

6.1. Test details

KGLS has been implemented in Java and all tests are performed within the Eclipse development environment on an AMD Ryzen 3 1300X CPU working at 3.5GHz on Windows 10, using a single thread. According to PassMark Software (2018) this setup is about 25% faster than the setup used to test HGSADC and ILS (Xeon CPU 3.07GHz). On each instance we run KGLS up to a defined time limit of $3\frac{N}{100}$ minutes, i.e., we allow 3 minutes of computation time per 100 customers. We observed in pre-tests that larger time limits yield only marginal improvements in solutions quality, whereas most solutions are found in significant less time. When we double the time limit, we observe an average improvement by 0.05% on the $\mathbb{U}$-instances. For each instance we then report the best solution found within this time horizon as well as the required computation time to obtain this solution. In the implementation each node is represented by an object which contains distance information $d(\cdot)$, $d^g(\cdot)$, and penalty information to all other nodes, pre-processed information about the C nearest nodes as well as the respective insertion costs, alongside basic information such as demand, its current route as well as the position in its current route. Routes are represented by a list of references to the respective nodes. For the encoding of routes, and also for other purposes such as the encoding of candidate moves, we used the datatype *ArrayList*, since it has a dynamic size and elements can be accessed in constant time. Even though insertions and deletions of elements require linear time (for instance to execute a move), they are executed not as often as accessing elements (for instance to evaluate a move).

[2]We want to remark that the average is just a rough estimate of an expected result. A statistical study on the performance of randomized algorithms proposed elsewhere is beyond the scope of this study.

Table 2: Results on the 100 𝕌-instances, as average over all instances and for instances with a certain number of customers N. The gap is reported in % with respect to the BKS, and the time is reported in minutes, based on Uchoa et al. (2017).

	ILS		HGSADC		KGLS	
N	Gap	Time	Gap	Time	Gap	Time
100-250	0.31%	2.4	0.07%	6.0	0.28%	1.8
250-500	0.53%	23.1	0.24%	30.3	0.59%	4.4
500-1000	0.72%	195.7	0.24%	268.6	0.45%	9.9
	0.53%	71.7	0.19%	98.8	0.44%	5.3

Table 3: Results on the 23 MDVRP ℂ-instances, as average over all instances. The gap is reported in % with respect to the BKS, and the time is reported in minutes, based on Vidal et al. (2013b).

ALNS		HGSADC		KGLS	
Gap	Time	Gap	Time	Gap	Time
0.40%	3.8	0.06%	4.1	0.08%	0.7

6.2. Results

The aggregated results on the VRP benchmark set for KGLS are presented in Table 2, detailed results per instance can be found in the appendix. We report the average gap in% with respect to the best known solutions (BKS) indicated in the respective papers, as well as the computation time in minutes.

We observe that KGLS computes competitive solutions within a 1% range of the best known solutions for almost all 𝕌-instances. On average the quality of the computed solutions are comparable to those of ILS and within a 0.25% range of those of HGSADC. The results in Table 3 highlight that KGLS also computes high-quality solutions for MDVRP instances, performing similarly to HGSADC. These competitive results are computed in short computing times, and especially on instances of larger size does KGLS only require a fraction of computation time in comparison to state-of-the-art heuristics. Generally, it does not require more than a few minutes to compute solutions with very small gaps for VRP as well as MDVRP instances with several hundred customers, resulting in an excellent time-quality trade-off. This time-quality trade-off can be attributed to a good scalability, and opens up the potential to further improve the heuristic. For instance, we observed that resets can further increase the quality of computed solutions. If KGLS failed to find improvements for some time, the current solution can be reset to the previously best found solution while all accumulated penalties are removed.

In Figure 10 (left) we visualize the scaling of the heuristic's computing time with growing instance size. KGLS computes solutions of similar quality, while the required computation time grows more or less as a linear function of the instance size. This

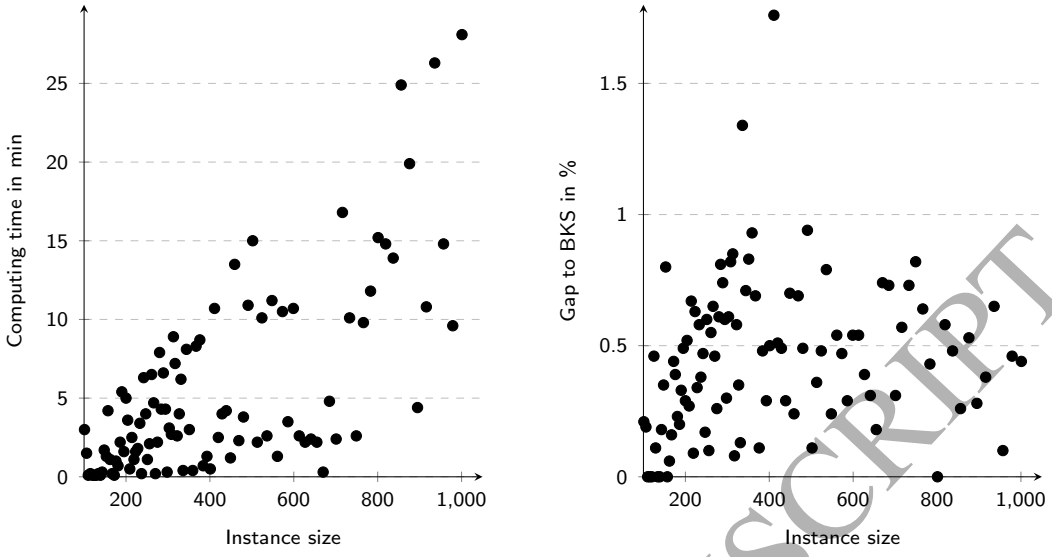


Figure 10: Computation time (left) versus performance (right) in dependence of instance size on the 100 𝕌-instances. Solutions of a similar high quality are found within computation times that grow approximately linear in the instance size.

linear scaling has its origin in the efficient design of the local search operators (see Section 2), which, despite all the heuristic pruning, generates high-quality solutions.

This efficient generation of a multitude of routing solutions also allows to solve MTVRP instances. The addition of simple bin-packing heuristics allows to validate that KGLS computes feasible solutions for 87 out of 99 MTVRP instances that are known to be solvable. For many of those instances KGLS finds the optimal solution, and on 21 instances it improves the BKS, challenging dedicated MTVRP heuristics. As a result, the average gap on those 87 instances to the BKS reported in Cattaruzza et al. (2016) is -0.06%, computed in 26 seconds (compared to a 0.68% gap of MA and to a 0.69% gap of ALNSP on those instance, obtained after similar computing times). We want to stress that the validity of such a comparison is limited, given that ALNPS and MA find feasible solutions for all instances (and are designed to do so). However, these results show the potential of applying an efficient local-search based heuristic to the MTVRP.

All in all, the presented KGLS consistently produces competitive solutions for a large variety of instances in short computation times.

7. Conclusions and future research

In this paper we have demonstrated how to build a heuristic for the VRP around a well-implemented local search. We implemented three complementary local search operators using ideas from sequential search and pruning. This local search was embedded in a guided local search framework that penalizes and removes bad edges. To detect bad edges more accurately, we used insights from an exploratory study on prop-

erties of VRP solutions. Thorough experimentation on all key components showed that relatively little perturbation combined with large neighborhoods results in an effective local search.

The resulting heuristic computes high-quality solutions for a wide range of benchmark instances in a few seconds or minutes. Therefore, it presents a suitable solution approach to routing problems that are restricted with respect to computation time. On the other hand, it might also be a starting point for further extensions to generate even better solutions, if more computation time is available. It can be extended by, for instance, adapting the edge penalization through learning mechanisms, or by embedding the heuristic into another metaheuristic. At the same time, we demonstrated that the heuristic can be used to solve other problem variants like the MDVRP and the MTVRP.

From a more general perspective, we have shown that the development of a successful heuristic for the VRP does not require new operators or complex frameworks. Using local search operators that have been proven to work well, paired with an efficient implementation and the utilization of problem-knowledge, is sufficient to create powerful heuristics that work well on a wide range of instances and problem variations. Rather than through the development of new heuristic operators, let alone new metaheuristic frameworks, algorithms can be improved by putting more research emphasis on how to use *existing* methods in a more efficient and intelligent way. The utilization of problem-specific knowledge is a promising starting point for this. Knowing the structural properties that distinguish a near-optimal solution from a suboptimal solution allows an algorithm to prioritize certain features in the search process, and to develop operators that tackle a problem more efficiently. This paper has presented one of the first approaches in this challenging research direction.

References

Applegate, D., Cook, W., and Rohe, A. (2003). Chained lin-kernighan for large traveling salesman problems. *INFORMS Journal on Computing*, 15(1):82–92.

Arnold, F. and Sörensen, K. (2018). What makes a VRP solution good? The generation of problem-specific knowledge for heuristics. *Computers & Operations Research*. Advance online publication. doi:10.1016/j.cor.2018.02.007.

Cattaruzza, D., Absi, N., and Feillet, D. (2016). Vehicle routing problems with multiple trips. *4OR*, 14(3):223–259.

Cattaruzza, D., Absi, N., Feillet, D., and Vidal, T. (2014). A memetic algorithm for the multi trip vehicle routing problem. *European Journal of Operational Research*, 236(3):833–848.

Clarke, G. and Wright, J. W. (1964). Scheduling of vehicles from a central depot to a number of delivery points. *Operations Research*, 12(4):568–581.

Cordeau, J.-F., Gendreau, M., and Laporte, G. (1997). A tabu search heuristic for periodic and multi-depot vehicle routing problems. *Networks*, 30(2):105–119.

Ergun, Ö., Orlin, J. B., and Steele-Feldman, A. (2006). Creating very large scale neighborhoods out of smaller ones by compounding moves. *Journal of Heuristics*, 12(1):115–140.

François, V., Arda, Y., Crama, Y., and Laporte, G. (2016). Large neighborhood search for multi-trip vehicle routing. *European Journal of Operational Research*, 255(2):422–441.

Gendreau, M., Hertz, A., and Laporte, G. (1992). New insertion and postoptimization procedures for the traveling salesman problem. *Operations Research*, 40(6):1086–1094.

Glover, F. (1996). Ejection chains, reference structures and alternating path methods for traveling salesman problems. *Discrete Applied Mathematics*, 65(1):223–253.

Glover, F. (2007). Tabu search—uncharted domains. *Annals of Operations Research*, 149(1):89–98.

Groër, C., Golden, B., and Wasil, E. (2010). A library of local search heuristics for the vehicle routing problem. *Mathematical Programming Computation*, 2(2):79–101.

Gutjahr, W. (2010). Stochastic search in metaheuristics. In Gendreau, M. and Potvin, J.-Y., editors, *Handbook of Metaheuristics (2nd ed.)*, volume 146 of *International Series in Operations Research & Management Science*, pages 573–597. Springer, New York.

Helsgaun, K. (2000). An effective implementation of the Lin–Kernighan traveling salesman heuristic. *European Journal of Operational Research*, 126(1):106–130.

Irnich, S., Funke, B., and Grünert, T. (2006). Sequential search and its application to vehicle-routing problems. *Computers & Operations Research*, 33(8):2405–2429.

Johnson, D. S., Papadimitriou, C. H., and Yannakakis, M. (1988). How easy is local search? *Journal of Computer and System Sciences*, 37(1):79–100.

Kilby, P., Prosser, P., and Shaw, P. (1999). Guided local search for the vehicle routing problem with time windows. In *Meta-heuristics*, pages 473–486. Springer.

Laporte, G. (2007). What you should know about the vehicle routing problem. *Naval Research Logistics (NRL)*, 54(8):811–819.

Lin, S. and Kernighan, B. W. (1973). An effective heuristic algorithm for the traveling-salesman problem. *Operations Research*, 21(2):498–516.

Mester, D. and Bräysy, O. (2007). Active-guided evolution strategies for large-scale capacitated vehicle routing problems. *Computers & Operations Research*, 34(10):2964–2975.

Mladenović, N. and Hansen, P. (1997). Variable neighborhood search. *Computers & Operations Research*, 24(11):1097–1100.

Montoya-Torres, J., Franco, J., Isaza, S., Jiménez, H., and Herazo-Padilla, N. (2015). A literature review on the vehicle routing problem with multiple depots. *Computers & Industrial Engineering*, 79:115–129.

Nagata, Y. and Bräysy, O. (2009). Edge assembly-based memetic algorithm for the capacitated vehicle routing problem. *Networks*, 54(4):205.

PassMark Software (2018). CPU benchmarks. `https://www.cpubenchmark.net/`. Accessed: 2018-02-05.

Pisinger, D. and Ropke, S. (2007). A general heuristic for vehicle routing problems. *Computers & Operations Research*, 34(8):2403–2435.

Rego, C. (2001). Node-ejection chains for the vehicle routing problem: Sequential and parallel algorithms. *Parallel Computing*, 27(3):201–222.

Salhi, S. (2014). Handbook of metaheuristics. *Journal of the Operational Research Society*, 65(2):320–320.

Simchi-Levi, D. (1994). New worst-case results for the bin-packing problem. *Naval Research Logistics (NRL)*, 41(4):579–585.

Sörensen, K., Sevaux, M., and Schittekat, P. (2008). "multiple neighbourhood" search in commercial vrp packages: Evolving towards self-adaptive methods. In *Adaptive and Multilevel Metaheuristics*, pages 239–253. Springer.

Subramanian, A., Uchoa, E., and Ochi, L. S. (2013). A hybrid algorithm for a class of vehicle routing problems. *Computers & Operations Research*, 40(10):2519–2531.

Taillard, É., Badeau, P., Gendreau, M., Guertin, F., and Potvin, J.-Y. (1997). A tabu search heuristic for the vehicle routing problem with soft time windows. *Transportation Science*, 31(2):170–186.

Taillard, E. D., Laporte, G., and Gendreau, M. (1996). Vehicle routeing with multiple use of vehicles. *Journal of the Operational Research Society*, 47(8):1065–1070.

Toth, P. and Vigo, D. (2003). The granular tabu search and its application to the vehicle-routing problem. *Informs Journal on Computing*, 15(4):333–346.

Uchoa, E., Pecin, D., Pessoa, A., Poggi, M., Vidal, T., and Subramanian, A. (2017). New benchmark instances for the capacitated vehicle routing problem. *European Journal of Operational Research*, 257(3):845–858.

Vidal, T., Crainic, T. G., Gendreau, M., and Prins, C. (2013a). Heuristics for multi-attribute vehicle routing problems: A survey and synthesis. *European Journal of Operational Research*, 231(1):1–21.

Vidal, T., Crainic, T. G., Gendreau, M., and Prins, C. (2013b). A hybrid genetic algorithm with adaptive diversity management for a large class of vehicle routing problems with time-windows. *Computers & Operations Research*, 40(1):475–489.

Voudouris, C. and Tsang, E. P. (2003). *Guided local search*. Springer.

Table 4: Results on the 𝕌-instances. The number of customers N is given in parentheses. The gap is reported in % with respect to the BKS, and the time is reported in minutes, based on Uchoa et al. (2017). The best solution per instance, as well as the shortest running time, is highlighted in gray.

Instance	ILS			HGSADC			KGLS		
	Value	Gap	Time	Value	Gap	Time	Value	Gap	Time
P01 (101)	27591.0	0.00	0.1	27591.0	0.00	1.4	27650	0.21	3.0
P02 (106)	26375.9	0.05	2.0	26381.8	0.08	4.0	26411	0.19	1.5
P03 (110)	14971.0	0.00	0.2	14971.0	0.00	1.6	14971	0.00	0.1
P04 (115)	12747.0	0.00	0.2	12747.0	0.00	1.8	12747	0.00	0.2
P05 (120)	13337.6	0.04	1.7	13332.0	0.00	2.3	13332	0.00	0.1
P06 (125)	55673.8	0.24	1.4	55542.1	0.01	2.7	55798	0.47	0.1
P07 (129)	28998.0	0.20	1.9	28948.5	0.03	2.7	28973	0.11	0.1
P08 (134)	10947.4	0.29	2.1	10934.9	0.17	3.3	10916	0.00	0.2
P09 (139)	13603.1	0.10	1.6	13590.0	0.00	2.3	13590	0.00	0.1
P10 (143)	15745.2	0.29	1.6	15700.2	0.00	3.1	15728	0.18	0.3
P11 (148)	43452.1	0.01	0.8	43448.0	0.00	3.2	43599	0.35	1.7
P12 (153)	21400.0	0.85	0.5	21226.3	0.03	5.5	21390	0.80	1.3
P13 (157)	16876.0	0.00	0.8	16876.0	0.00	3.2	16876	0.00	4.2
P14 (162)	14160.1	0.16	0.5	14141.3	0.02	3.3	14147	0.06	1.1
P15 (167)	20608.7	0.25	0.9	20563.2	0.03	3.7	20589	0.16	0.2
P16 (172)	45616.1	0.02	0.6	45607.0	0.00	3.8	45807	0.44	0.1
P17 (176)	48249.8	0.92	1.1	47957.2	0.30	7.6	47998	0.39	1.0
P18 (181)	25571.5	0.01	1.6	25591.1	0.09	6.3	25628	0.23	0.7
P19 (186)	24186.0	0.17	1.7	24147.2	0.01	5.9	24194	0.20	2.2
P20 (190)	17143.1	0.96	2.1	16987.9	0.05	12.1	17036	0.33	5.4
P21 (195)	44234.3	0.02	0.9	44244.1	0.04	6.1	44442	0.49	1.6
P22 (200)	58697.2	0.20	7.5	58626.4	0.08	8.0	58747	0.29	5.0
P23 (204)	19625.2	0.31	1.1	19571.5	0.03	5.4	19666	0.52	3.6
P24 (209)	30765.4	0.36	3.8	30680.4	0.08	8.6	30738	0.27	0.5
P25 (214)	11126.9	0.25	2.3	10877.4	0.20	10.2	10929	0.67	2.5
P26 (219)	117595.0	0.00	0.9	117604.9	0.01	7.7	117696	0.09	1.1
P27 (223)	40533.5	0.24	8.5	40499.0	0.15	8.3	40691	0.63	1.6
P28 (228)	25795.8	0.21	2.4	25779.3	0.14	9.8	25830	0.34	1.8
P29 (233)	19336.7	0.55	3.0	19288.4	0.30	6.8	19341	0.58	3.4
P30 (237)	27078.8	0.14	3.5	27067.3	0.09	8.9	27146	0.38	0.2
P31 (242)	82874.2	0.15	17.8	82948.7	0.24	12.4	83144	0.47	6.3
P32 (247)	37507.2	0.63	2.1	37284.4	0.03	20.4	37336	0.17	4.0
		0.31	2.4		0.07	6.0		0.28	1.8

Table 5: Results on the 𝕌-instances. The number of customers N is given in parentheses. The gap is reported in % with respect to the BKS, and the time is reported in minutes, based on Uchoa et al. (2017). The best solution per instance, as well as the shortest running time, are highlighted in gray.

Instance	ILS			HGSADC			KGLS		
	Value	Gap	Time	Value	Gap	Time	Value	Gap	Time
P33 (251)	38840.0	0.40	10.8	38796.4	0.29	11.7	38916	0.60	1.1
P34 (256)	18883.9	0.02	2.0	18880.0	0.00	6.5	18899	0.10	2.1
P35 (261)	26869.0	1.17	6.7	26629.6	0.27	12.7	26704	0.55	6.5
P36 (266)	75563.3	0.11	10.0	75759.3	0.37	21.4	75966	0.65	4.7
P37 (270)	35363.4	0.21	9.1	35367.2	0.22	11.3	35453	0.46	0.2
P38 (275)	21256.0	0.05	3.6	21280.6	0.17	12.0	21300	0.26	2.2
P39 (280)	33769.4	0.80	9.6	33605.8	0.31	19.1	33709	0.61	7.9
P40 (284)	20448.5	1.10	8.6	20286.4	0.30	19.9	20389	0.81	4.3
P41 (289)	95450.6	0.28	16.1	95469.5	0.30	21.3	95885	0.74	6.6
P42 (294)	47254.7	0.19	12.4	47259.0	0.20	14.7	47450	0.60	4.3
P43 (298)	34356.0	0.37	6.9	34292.1	0.18	10.9	34332	0.30	0.3
P44 (303)	21895.8	0.69	14.2	21850.9	0.49	17.3	21877	0.61	3.1
P45 (308)	26101.1	0.94	9.5	25895.4	0.14	15.3	26072	0.82	2.7
P46 (313)	94297.3	0.27	17.5	94265.2	0.24	22.4	94844	0.85	8.9
P47 (317)	78356.0	0.00	8.6	78387.8	0.04	22.4	78414	0.08	7.2
P48 (322)	29991.3	0.42	14.7	29956.1	0.30	15.2	30038	0.58	2.6
P49 (327)	27812.4	0.93	19.1	27628.2	0.26	18.2	27652	0.35	4.0
P50 (331)	31235.5	0.43	15.7	31159.6	0.18	24.4	31142	0.13	6.2
P51 (336)	139461.0	0.19	21.4	139534.9	0.24	38.0	141060	1.34	0.4
P52 (344)	42284.0	0.44	22.6	42208.8	0.26	21.7	42398	0.71	8.1
P53 (351)	26150.3	0.79	25.2	26014.0	0.26	33.7	26162	0.83	3.0
P54 (359)	52076.5	1.10	48.9	51721.7	0.41	34.9	51988	0.93	0.4
P55 (367)	23003.2	0.83	13.1	22838.4	0.11	22.0	22972	0.69	8.3
P56 (376)	147713.0	0.00	7.1	147750.2	0.03	28.3	147879	0.11	8.7
P57 (384)	66372.5	0.44	34.5	66270.2	0.29	40.2	66400	0.48	0.7
P58 (393)	38457.4	0.49	20.8	38374.9	0.28	28.7	38381	0.29	1.3
P59 (401)	66715.1	0.71	60.4	66365.4	0.18	49.5	66571	0.50	0.5
P60 (411)	19954.9	1.20	23.8	19743.8	0.13	34.7	20065	1.76	10.7
P61 (420)	107838.0	0.04	22.2	107924.1	0.12	53.2	108351	0.51	2.5
P62 (429)	65746.6	0.37	38.2	65648.5	0.23	41.5	65820	0.49	4.0
P63 (439)	36441.6	0.13	39.6	36451.1	0.15	34.6	36502	0.29	4.2
P64 (449)	56204.9	1.53	59.9	55553.1	0.35	64.9	55747	0.70	1.2
P65 (459)	24462.4	1.16	60.6	24272.6	0.38	42.8	24240	0.24	13.5
P66 (469)	222182.0	0.12	36.3	222617.1	0.32	86.7	223433	0.69	2.3
P67 (480)	89871.2	0.38	50.4	89760.1	0.25	67.0	89970	0.49	3.8
P68 (491)	67226.7	0.89	52.2	66898.0	0.40	71.9	67261	0.94	10.9
		0.53	23.1		0.24	30.3		0.59	4.4

Table 6: Results on the 𝕌-instances. The number of customers N is given in parentheses. The gap is reported in % with respect to the BKS, and the time is reported in minutes, based on Uchoa et al. (2017). The best solution per instance, as well as the shortest running time, is highlighted in gray.

Instance	ILS			HGSADC			KGLS		
	Value	Gap	Time	Value	Gap	Time	Value	Gap	Time
P69 (502)	69346.8	0.14	80.8	69328.8	0.11	63.6	69327	0.11	15.0
P70 (513)	24434.0	0.96	35.0	24296.6	0.40	33.1	24287	0.36	2.2
P71 (524)	155005.0	0.27	27.3	154979.5	0.25	80.7	155342	0.48	10.1
P72 (536)	95700.7	0.61	62.1	95330.6	0.22	107.5	95875	0.79	2.6
P73 (548)	86874.1	0.19	64.0	86998.5	0.33	84.2	86920	0.24	11.2
P74 (561)	43131.3	0.88	68.9	42866.4	0.26	60.6	42989	0.54	1.3
P75 (573)	51173.0	0.77	112.0	50915.1	0.27	188.2	51020	0.47	10.5
P76 (586)	190919.0	0.20	78.5	190838.0	0.15	175.3	191094	0.29	3.5
P77 (599)	109384.0	0.52	73.0	109064.2	0.23	125.9	109397	0.54	10.7
P78 (613)	60444.2	1.11	74.8	59960.0	0.30	117.3	60100	0.54	2.6
P79 (627)	62905.6	0.87	162.7	62524.1	0.25	239.7	62612	0.39	2.2
P80 (641)	64606.1	1.20	140.4	64192.0	0.55	158.8	64035	0.31	2.4
P81 (655)	106782.0	0.00	47.2	106899.1	0.11	150.5	106969	0.18	2.2
P82 (670)	147676.0	0.66	61.2	147222.7	0.35	264.1	147796	0.74	0.3
P83 (685)	68988.2	0.82	73.9	68654.1	0.33	156.7	68927	0.73	4.8
P84 (701)	83042.2	0.91	210.1	82487.4	0.24	253.2	82551	0.31	2.4
P85 (716)	44171.6	1.49	225.8	43641.4	0.27	264.3	43772	0.57	16.8
P86 (733)	137045.0	0.50	111.6	136587.6	0.16	244.5	137364	0.73	10.1
P87 (749)	78275.9	0.74	127.2	77864.9	0.21	313.9	78337	0.82	2.6
P88 (766)	115738.0	0.92	242.1	115147.9	0.41	383.0	115418	0.64	9.8
P89 (783)	73722.9	1.37	235.5	73009.6	0.39	269.7	73038	0.43	11.8
P90 (801)	74005.7	0.57	432.6	73731.0	0.20	289.2	73525	-0.08	15.2
P91 (819)	159425.0	0.51	148.9	158899.3	0.18	374.3	159525	0.58	14.8
P92 (837)	195027.0	0.39	173.2	194476.5	0.11	463.4	195203	0.48	13.9
P93 (856)	89277.6	0.24	153.7	89238.7	0.20	288.4	89289	0.26	24.9
P94 (876)	100417.0	0.70	409.3	99884.1	0.17	495.4	100244	0.53	19.9
P95 (895)	54958.5	1.45	410.2	54439.8	0.49	321.9	54321	0.28	4.4
P96 (916)	330948.0	0.33	226.1	330198.3	0.11	560.8	331081	0.38	10.8
P97 (936)	134530.0	1.07	202.5	133512.9	0.31	531.5	133968	0.65	26.3
P98 (957)	85936.6	0.31	311.2	85822.6	0.18	432.9	85756	0.10	14.7
P99 (979)	120253.0	0.89	687.2	119502.1	0.26	554.0	119737	0.46	9.6
P100 (1001)	73985.4	1.71	792.8	72956.0	0.29	549.0	73060	0.44	28.1
		0.72	195.7		0.24	268.6		0.45	9.9

Table 7: Results on the $\mathbb{C}$-instances. The number of customers N is given in parentheses, and D denotes the number of depots. The gap is reported in % with respect to the BKS, and the time is reported in minutes, based on Cordeau et al. (1997). The best solution per instance, as well as the shortest running time, is highlighted in gray.

Instance	D	ALNS			HGSADC			KGLS		
		Value	Gap	Time	Value	Gap	Time	Value	Gap	Time
P01 (50)	4	576.87	0.00	0.5	576.87	0.00	0.2	576.87	0.00	0.1
P02 (50)	4	473.53	0.00	0.5	473.53	0.00	0.2	473.53	0.00	0.1
P03 (75)	2	641.19	0.00	1.1	641.19	0.00	0.4	641.19	0.00	0.2
P04 (100)	2	1006.9	0.49	1.5	1001.23	0.00	1.9	1001.06	0.00	1.0
P05 (100)	2	752.3	0.31	2	750.03	0.00	1.1	750.59	0.07	0.9
P06 (100)	3	883.01	0.74	1.6	876.50	0.00	1.1	876.70	0.02	0.9
P07 (100)	4	889.36	0.84	1.5	884.43	0.28	1.6	883.91	0.22	1.4
P08 (249)	2	4421.03	1.10	5.6	4397.42	0.56	10.0	4390.22	0.40	0.1
P09 (249)	3	3892.50	0.88	6.0	3868.59	0.26	9.5	3866.95	0.21	1.7
P10 (249)	4	3666.85	0.98	6.1	3636.08	0.14	9.8	3658.64	0.76	0.2
P11 (249)	4	3573.23	0.77	6.0	3548.25	0.06	7.1	3550.88	0.14	4.0
P12 (80)	2	1319.13	0.01	1.3	1318.95	0.00	0.5	1318.95	0.00	0.1
P13 (80)	2	1318.95	0.00	1.0	1318.95	0.00	0.6	1318.95	0.00	0.1
P14 (80)	2	1360.12	0.00	1.0	1360.12	0.00	0.6	1360.12	0.00	0.1
P15 (160)	4	2519.64	0.57	4.2	2505.42	0.00	1.9	2505.42	0.00	0.1
P16 (160)	4	2573.95	0.07	3.1	2572.23	0.00	2.0	2572.23	0.00	0.1
P17 (160)	4	2709.09	0.00	3.0	2709.09	0.00	2.1	2709.09	0.00	0.1
P18 (240)	6	3736.53	0.91	7.0	3702.85	0.00	4.5	3702.85	0.00	3.4
P19 (240)	6	3838.76	0.31	5.3	3827.06	0.00	4.2	3827.06	0.00	0.0
P20 (240)	6	4064.76	0.16	5.0	4058.07	0.00	4.4	4058.07	0.00	0.3
P21 (360)	6	5501.58	0.46	9.7	5476.41	0.00	10.0	5482.47	0.11	1.0
P22 (360)	6	5722.19	0.35	7.7	5702.16	0.00	10.0	5702.16	0.00	0.1
P23 (360)	6	6092.66	0.23	7.4	6078.75	0.00	10.0	6078.75	0.00	1.0
			0.40	3.8		0.06	4.1		0.08	0.7

Table 8: Results on the MTVRP instances by Taillard et al. (1996). The number of customers N is given in parentheses. Each instance is defined by the number of vehicles M and a time horizon T^1 or T^2. The gap is reported in % with respect to the BKS in Cattaruzza et al. (2016), and the time is reported in seconds. NF denotes that no feasible solution could be found.

			KGLS				KGLS		
Instance	M	T^1	Value	Gap	Time	T^2	Value	Gap	Time
CMT1 (50)	1	551	524.61	0.00	1	577	524.61	0.00	1
	2	275	533.00	0.00	25	289	529.85	0.00	1
	3					192	NF		
	4					144	NF		
CMT2 (75)	1	877	835.26	0.00	1	919	835.26	0.00	1
	2	439	835.26	0.00	1	459	835.26	0.00	1
	3	292	835.26	0.00	1	306	835.26	0.00	1
	4	219	835.26	0.00	1	230	835.26	0.00	1
	5	175	836.71	0.11	1	184	835.26	0.00	1
	6	146	NF			153	847.7	1.01	13
	7					131	NF		
CMT3 (100)	1	867	826.14	0.00	3	909	826.14	0.00	3
	2	434	826.14	0.00	3	454	826.14	0.00	3
	3	289	826.14	0.00	3	303	826.14	0.00	3
	4	217	828.73	-0.09	30	227	826.14	0.00	3
	5	173	NF			182	832.88	0.06	2
	6	145	NF			151	837.34	0.36	1
CMT4 (150)	1	1080	1031.07	0.01	4	1131	1031.07	0.00	4
	2	540	1031.07	0.00	16	566	1031.07	0.06	16
	3	360	1031.07	0.26	4	377	1031.07	-0.05	8
	4	270	1031.07	0.00	4	283	1031.07	0.00	4
	5	216	1031.07	0.00	4	226	1031.07	0.02	4
	6	180	1038.05	0.33	159	189	1031.07	0.06	4
	7	154	NF			162	1035.41	-0.06	111
	8	135	NF			141	NF		
CMT5 (199)	1	1356	1297.70	-0.36	24	1421	1297.70	-0.17	25
	2	678	1297.70	-0.34	24	710	1297.70	-0.59	25
	3	452	1297.70	-0.29	24	474	1297.70	-0.26	25
	4	339	1297.70	-0.54	24	355	1297.70	-0.46	25
	5	271	1298.07	-0.15	59	284	1297.70	-0.45	25
	6	226	1297.70	-0.43	25	237	1297.70	-0.65	25
	7	194	1298.50	-0.83	60	203	1297.70	-0.29	25
	8	170	1296.02	-0.61	35	178	1297.70	-0.85	25
	9	151	1298.04	-0.76	48	158	1294.85	-0.95	35
	10	136	NF			142	1310.04	0.09	9
CMT11 (120)	1	1094	1042.64	0.05	87	1146	1042.64	0.05	84
	2	547	1042.64	0.05	87	573	1042.64	0.05	84
	3	365	1042.64	0.05	87	382	1042.64	0.05	84
	4	274	NF			287	1042.51	0.04	106
	5	219	1044.51	0.23	60	229	1043.80	0.16	119
CMT12 (100)	1	861	819.56	0.00	1	902	819.56	0.00	1
	2	430	819.56	0.00	1	451	819.56	0.00	1
	3	287	819.56	0.00	1	301	819.56	0.00	1
	4	215	819.56	0.00	1	225	819.56	0.00	1
	5	172	NF			180	824.78	0.00	1
	6					150	823.14	0.00	56
F11 (71)	1	254	241.97	0.00	81	266	241.97	0.00	1
	2	127	252,27	0.57	26	133	241.97	0.00	1
	3			0.05	87	89	254,07	0.00	1
F12 (134)	1	1221	1162.96	0.00	40	1279	1162.96	0.00	40
	2	611	1162.96	0.00	40	640	1162.96	0.00	40
	3	407	1162.96	0.00	40	426	1162.96	0.00	40