DEPARTMENT OF ENGINEERING MANAGEMENT

A simple, deterministic, and efficient knowledge-driven heuristic for the vehicle routing problem

Florian Arnold & Kenneth Sörensen

DEPARTMENT OF ENGINEERING MANAGEMENT

A simple, deterministic, and efficient knowledge-driven heuristic for the vehicle routing problem

Florian Arnold & Kenneth Sörensen

RESEARCH PAPER 2017-012
DECEMBER 2017

D/2017/1169/012

A simple, deterministic, and efficient knowledge-driven heuristic 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

In this paper we develop a heuristic for the capacitated vehicle routing problem that revolves around three complementary local search operators, embedded in a guided local search framework. The efficiency of the operators is guaranteed by using knowledge, obtained through data mining, on the attributes of undesirable edges. In spite of its straightforward design and the fact that it is completely deterministic, the heuristic is competitive with the best heuristics in the literature in terms of accuracy and speed. Moreover, it can be readily extended to solve a wide range of vehicle routing problems, which we demonstrate by applying it to the multi-depot vehicle routing problem.

Keywords: vehicle routing problems, heuristics, metaheuristics

1. Introduction

The vehicle routing problem (VRP) is one of the most intensively studied problems in the field of combinatorial optimization. Over the last decades, this challenging optimization problem has yielded thousands of publications and a plethora of innovative optimization techniques.

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 distance traveled by all vehicles combined is minimal. Customers have a known demand, and the total demand served by each vehicle cannot exceed its capacity. 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 over the last years, 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

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

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

The success of heuristics has triggered a race for ever better and faster algorithms. State-of-the-art heuristics can solve instances of several hundred customers to near-optimality in a few minutes of computing time (Vidal et al., 2013a). However, the competitive nature of the research field has resulted in an exclusive focus in the literature on accuracy (the ability to find good solutions) and speed (the ability to find solutions quickly), disregarding other desirable properties such as simplicity and flexibility, defined in Cordeau et al. (2002). As a consequence, many state-of-the-art heuristics are very complicated, both in terms of their design and in the number of parameter that need to be set. This complexity leads to two disadvantages. First, studying the impact of the individual components and their parameters, which in turn allows to generalize the findings and generate a deeper understanding of why exactly the algorithm works well, becomes more difficult. Secondly, re-implementing heuristics (either to validate their results or to use them in other research or even in practice) becomes a near-impossible task. Not only does this limit the practical usability of most heuristics for the VRP, it also raises questions about the reproducibility of the presented results.

The challenge raised in this paper is whether it is possible to develop a heuristic that matches the performance of the best-performing heuristics in the literature in terms of accuracy and speed, but also satisfies the simplicity criterion in that it is easy to understand and to study. Additionally, the heuristic should preferably be flexible, i.e., it should be readily applicable to a variety of routing problems.

Empirical evidence shows that high-quality local search operators function as the key optimization engine of most, if not all, successful heuristics (see also Rego and Glover (2007) for the TSP). Over the years, numerous local search operators have been designed that vary in complexity and application area.

In this paper we aim to demonstrate that a limited number of well-chosen and well-implemented local search operators suffice to create a heuristic that can compete with the best heuristics in the literature. To this end we combine three of the most powerful local search techniques, and implement them in a very 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 matches the performance of the best heuristics in the literature, it is also straightforward in its design and does not contain any components of which the contribution is unclear. Finally, we demonstrate the heuristic's flexibility by applying it to the VRP with multiple depots (MDVRP).

We motivate and introduce our local search components in Section 2. In Section 3 we demonstrate how local search can be guided by the penalization of edges. The complete heuristic is presented in Section 4 and analyzed in Section 5. In Section 6 we extend the heuristic to solve the MDVRP, and conduct detailed testing on the CVRP and MDVRP in Section 7. We conclude with a brief summary of our findings in Section 8.

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, i.e., the set of solutions that can be reached from the current solution by applying a single move of that type. Table 1 presents a brief summary of the most frequently used local search operators (a more comprehensive overview can be found in Vidal et al. (2013a)).

Table 1: Overview of popular local search operators (n denotes the number of customers)

Operator	Complexity	Description
2-opt	$O(n^2)$	Remove and reinsert 2 edges between customers in the same route
3-opt	$O(n^3)$	Remove and reinsert 3 edges between customers in the same route
Insert / Relocate	$O(n^2)$	Remove a customer and reinsert it in another position
Swap	$O(n^2)$	Exchange the position of two customers
Crossover	$O(n^2)$	Exchange the route ends (one or a sequence of customers) between two routes.
CROSS-exchange	$O(n^4)$	Exchange any two customer sequences between two routes.

After generating the neighborhood of the current solution, local search uses a move *strategy* to select at most one solution from the neighborhood 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. Even though, local optima are generally easier to find than global optima, it has been shown that the complexity of finding local optima relates to NP-completeness (Johnson et al., 1988).

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 explored. 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 solution quality (or probability of improvement).

This trade-off presents one of the greatest challenges in the design of an efficient local search algorithm. 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 'swap' can be easily implemented and quickly executed. However, they might not provide sufficient options to find improvements. Reversely, operators with large neighborhoods, such as CROSS-exchange generate a large number of neighbors, but are impractical for larger instances because of their high complexity.

Of course, the efficiency with which the respective local search operators can be implemented is equally important. Smart data structures, especially when combined with preprocessing and storage of auxiliary information, can reduce the number of required computational steps significantly. The complexity of local search operators can also be restricted by only considering promising neighbors, instead of generating the entire neighborhood, an approach which is called *heuristic pruning.* A metaheuristic that drastically restricts neighborhoods is, e.g., *granular tabu search* (Toth and Vigo, 2003).

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). Many, if not most, algorithms for the vehicle routing problem implement a large number of different local search operators, usually without investigating whether all operators have an effect on the quality of the solutions found. The active-guided evolution strategies of Mester and Bräysy (2007), e.g., use relocate, swap, 2-opt and Or-exchange (a form of relocate), 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-opt and CROSS-exchange. The genetic algorithm of Prins (2004) even uses nine different move types.

In summary, we argue that successful algorithms for the vehicle routing problem must use a small yet diverse set of well-chosen local search operators that have sufficiently large neighborhoods, but are efficiently implemented with heuristic pruning, i.e., are able to find improvements without exploring the neighborhood exhaustively.

Based on these observations, we design a powerful and efficient algorithm for the vehicle routing problem that is based entirely on a three well-implemented local search operators: CROSS-exchange, ejection chain, and the Lin-Kernighan heuristic. The motivation for this choice of operators is the following. Since CROSS-exchange incorporates the operators insert, swap, and crossover, this operator can be seen as an efficient generic operator to optimize a pair of routes, assuming that its computational complexity can be reduced. Improving only two routes at a time might not always be sufficient, especially when the considered routes are tightly constrained. In this case, we need a local search operator that can trigger changes in three or more routes. The ejection chain operator (EC) constitutes a powerful tool for this purpose. By performing a chain of moves, it can theoretically induce changes in all routes simultaneously. Both the CE and the EC operator perform operations on several routes, and, thus, they focus on *inter-route optimization.* Since routes should also be optimal in themselves, we

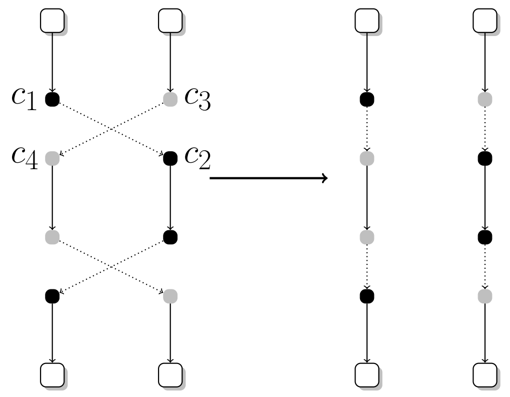


Figure 1: Illustration of the CROSS-exchange with sequences of two customers.

also need an operator that takes care of *intra-route optimization*. Usually, this is done by 2-opt or 3-opt moves (for instance in the heuristics mentioned above), which might not be sufficient when optimizing larger routes. Since the optimization of a single route corresponds to solving a travelling salesman problem (TSP), we choose one of the most efficient algorithms for the TSP, the Lin-Kernighan heuristic (LK).

In summary, we select a set of three complementary local search operators, CROSS-exchange, ejection chain, and Lin-Kernighan heuristic, each with proven effectiveness. In combination, these operators include all operators in Table 1. However, each of these operators generates a large neighborhood at the cost of high computational complexity. In the following, we show how to circumvent this issue and obtain a highly efficient local search algorithm.

2.1. CROSS-exchange

The CROSS-exchange operator (CE) is a generic local search operator that tries to make changes in two routes simultaneously (Taillard et al., 1997). Given a pair of routes, CE attempts to exchange any pair of (possibly empty) sequences of consecutive customers in both routes. An example of such a move is visualized in Fig. 1. CE generates a large neighborhood, which includes those of other popular operators like insert, swap, and crossover. If one sequence is empty and the other contains one customer, it performs an insert. It executes a swap if both sequences contain one customer. Likewise, if both sequences start at the depot and are not empty, it performs a crossover. Since CE tries to exchange any sequence in the first route with any sequence in the second route, and sequences can start and end at any customer within the route, it has a high computational complexity of $O(n^4)$. Therefore, the operator can only be used efficiently if its neighborhood is pruned and not all neighbor solutions are explored exhaustively.

The number of neighbors can be reduced significantly by only looking for exchanges in specific parts of the solution. For instance, assume we know that a certain route has a high potential for improvement, then we could limit the search to exchanges in which one sequence originates from this route. If we would even know which edge we have to remove, then we only need to consider exchanges in which one sequence starts from one of the adjacent nodes. For a CROSS-exchange, both of these nodes have to be connected to two adjacent nodes in another route. We can limit the number of potential candidates to those nodes, that are closest, a typical prunning approach that is used in many heuristics (e.g., in Mester and Bräysy (2005)).

Consider Fig. 1 for an example. We have identified (c_1, c_2) as an edge we want to remove (we explain how we do so in Section 3). In a preprocessing step, we have generated a list of nodes that are closest to c_1 and c_2, and now we evaluate, whether any of those is a promising candidate for a CE operation. As an example, let c_4 be be a node that is very close to c_1, and we try to add edge (c_1, c_4). Edge c_4 is a promising candidate, if the removal of (c_1, c_2) and (c_3, c_4) and, e.g., the addition of edges (c_1, c_4) and (c_2, c_3) (or (c_1, c_3) and (c_2, c_4)) results in an improvement. All of the above steps run in linear time.

The remaining task is to find sequences starting from the defined nodes c_2 and c_4, such that an exchange of these sequences results in a feasible solution with an improved solution value. The enumeration of all possible sequences has a complexity of $O(n^2)$ (note that one sequence can also be empty). However, many sequences will violate constraints, e.g., resulting in one of the routes becoming too long or exceeding the capacity of a vehicle. Those infeasible exchanges can be detected early and do not need to be considered further. As a result, we obtain an operator with a worst case complexity of $O(n^2)$, but with an actual performance that is closer to linear time.

2.2. Ejection chain

The ejection chain operator (EC), introduced in Glover (1996) and formalized in Rego (2001), considers changes in multiple routes simultaneously. Additionally, EC is very efficient in removing smaller routes and also creating new ones (Rousseau et al., 2002). It thereby complements the above introduced CE operator, which is limited to pairwise route optimization, and which is also more inflexible when it comes to the elimination of routes.

EC tries to relocate several customers in one sweep. Starting with a relocation of a customer from route a into route b, it then tries to relocate another customer from route b into route c (where $c = a$ is possible, or c is an empty route), and so on. In other words, it builds a chain of relocations. An example is visualized in Fig. 2.

If we limit the number of relocations per chain by l, then the complexity of EC is $O(n^{2l})$. For each relocation, we have to select a customer (any customer at the start, and otherwise any customer from the route into which the last customer was relocated) and a new position in a different route for this customer. Both these operations have a complexity of $O(n)$, and have to be executed for each of the l relocations. This high computational complexity can be reduced by applying heuristic prunning as well as preprocessing techniques.

As above, we assume that we know which edge to remove ((c_1^-, c_1) in Fig. 2). Having selected one edge to remove we only need to generate chains that start with a relocation of one of its adjacent customers (c_1^- or c_1). To limit the number of potential relocation

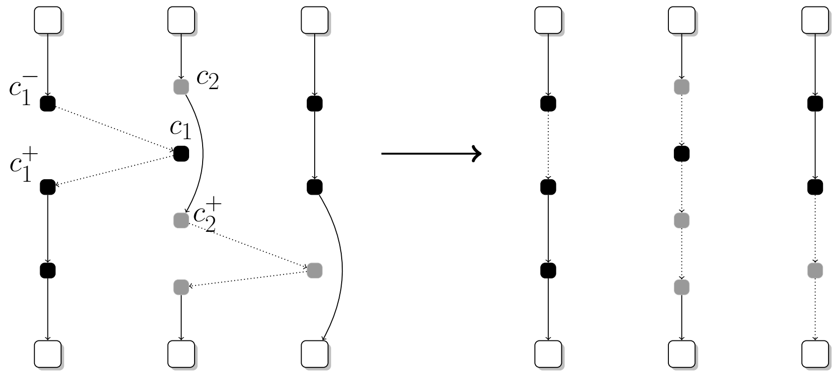


Figure 2: Illustration of the ejection chain with two relocations.

destinations, we only consider insertions next to close nodes (which are preprocessed as for the CE operator). In the example we try to insert c_1 next to c_2. The potential savings of this relocation can be preprocessed as well: the costs of detours and insertions are computed at the start, and only updated when necessary. More precisely, the savings of removing a customer c_1 from route a (detour) and relocating it next to customer c_2 in route b (we denote the neighbors of a node c_1 with c_1^+ and c_1^-, and c denotes the cost of an edge) amount to:

$$s(c_1, c_2) = c(c_1, c_1^+) + c(c_1, c_1^-) - c(c_1^+, c_1^-) + c(c_2, c_2^+) - c(c_1, c_2) - c(c_1, c_2^+) \qquad (1)$$

The relocation is executed if it has a positive saving, in which case we then try to add another relocation, starting from route b. For any customer in route b, we try to find a relocation such that the sum of the savings of all relocations remains positive, and route b fulfills all constraints. Consequently, for each relocation there might be numerous other relocations that continue the same chain, so that we obtain a tree structure (where each path from a leaf to the root represents one chain). This process is iterated until we have reached a certain depth, i.e., each considered chain has a maximal number of relocations l. The chain which realizes the highest savings and does not violate any constraints is then executed, or rather, all underlying relocations are executed.

All steps in this process run in linear time, except for the continuation of existing chains, where each customer of the current route is a candidate. Thus, we have a theoretical complexity of $O(n^{l-1})$. Since we only consider valid chains with positive savings, the runtime behaviour is usually much better in practice.

2.3. Lin-Kernighan Heuristic

The optimization of a VRP involves allocating customers to routes (inter-route-optimization), but also requires optimizing the routes in themselves (intra-route-optimization). EC and CE provide tools to tackle the first task, while the latter task is usually carried out by 2-opt moves. However, 2-opt or even 3-opt might not always result in optimal routes, especially if the routes are long.

We quantified this effect in a a simple experiment, in which we generated random TSP instances with random initial solutions, and compared the optimal solutions (computed with CPLEX) to the solutions obtained by iteratively using 2-opt and 3-opt moves. For routes with 10 customers, an average optimality gap of 0.5% was obtained, which increased to a considerable 2% for routes with 20 customers. Thus, there is need for a more powerful intra-route optimization, especially when faced with routes with many customers.

Optimizing the distance traveled in a single route of a vehicle routing problem corresponds to solving a traveling salesperson problem (TSP). Probably the most popular and efficient algorithm for the TSP is the Lin-Kernighan heuristic (LK) (Lin and Kernighan, 1973; Helsgaun, 2000). In simple terms, LK tries to find k-opt moves that improve the current solution. Since the search effort grows exponentially with increasing k, LK tries to reduce this complexity and only considers $k+1$-opt moves, if it has previously found an improvement with k-opt. More concretely, it tries to find improvements with 2-opt, and only continues with 3-opt if an improving move can be found. In this manner, it continues until an upper bound for k is reached. After each such iteration, the move with the most improvement is executed, and the next iteration starts with 2-opt again until no more improvements can be found.

The theoretical complexity of LK is quite high (O^k). However, our algorithm only uses LK at the start of the search and whenever changes are made with either EC or CE. Since the changes are usually quite small, LK will usually abort its search after 2-opt, since improvements with higher-order k-opt moves are unlikely to be found.

3. Guiding local search with problem-specific knowledge

The efficiency of the local search operators CE and EC hinges on the assumption that the algorithm knows which edge to remove from the current solution. Once we have identified such a 'bad' edge, we can use it as a starting point to efficiently use both EC and CE.

To increase the probability of CE and EC to start from a 'bad' edge, our algorithm penalizes such edges by increasing their cost value. Penalizing, and thereby removing, those features of a solution that are considered bad is the essential idea behind the guided local search (GLS) metaheuristic (Voudouris and Tsang, 2003). Even though GLS is not as popular as other heuristic concepts in local search (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).

In the context of the VRP, we penalize edges between customer pairs. More formally, we change the cost $c(i,j)$ of an edge in the current solution between customers i and j to c^g:

$$c^g(i,j) = c(i,j) + \lambda p(i,j) c(i,j), \tag{2}$$

where $p(i,j)$ counts the number of penalties of edge (i,j), λ controls the impact of penalties (usually with $\lambda \in [0.1, 0.3]$ (Kilby et al., 1999)).

A crucial question that needs to be answered is which edges to penalize, i.e., which edges should be considered detrimental to the quality of a solution. 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, but can be found in Arnold and Sörensen (2017).

In a nutshell, the methodology was as follows. 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 then computed an optimal or near-optimal solution, as well as a non-optimal solution. The non-optimal solutions were generated to have a pre-defined gap to the (near-)optimal value of either 2% or 4%. We then defined several metrics to characterize the solutions, i.e. 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 characterized about 192.000 solutions for 96.000 different VRP instances. We then used a classification learner to discover features that distinguish (near-)optimal from non-optimal solutions. A classification learner uses a certain percentage of randomly selected datapoints (solutions in the training set) to train an internal predictive model which tries to distinguish between optimal and non-optimal solutions. This predictive model is then validated with the help of the remaining datapoints (validation set). The percentage of correctly classified datapoints in the validation set reflects the prediction accuracy of the predictive model. Since, in our experiment, the number of datapoints associated with (near-)optimal solutions was the same as the one for non-optimal solutions, a prediction accuracy of higher than 50% indicates that predictive pattern within the data exist. As the most important result, we found a high predictive power in the defined metrics for all types of instances, and we were able to predict whether a solution belongs to either the class of (near-)optimal or the class of non-optimal solutions with an accuracy of up to 93%.

We also investigated which individual metrics were most predictive in distinguishing solutions of different quality. The results demonstrated that the most predictive metrics were the width and compactness of routes, i.e., solutions of higher quality tend to have narrower and more compact routes. 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 the customers 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

Table 2: Metrics with the highest individual prediction accuracy to distinguish between optimal and non-optimal solutions (as average over all experiments). W - Width, C - Compactness, I - Intersections, L - Length of edges connected to depot, D - Depth. The numbers present the prediction accuracy.

	2% gap					4% gap				
n	W	C	I	L	D	W	C	I	L	D
20-50	56.3	55.6	54.9	53.6	50.8	59.0	59.3	58.4	55.6	52.3
70-100	70.6	69.6	60.1	59.8	52.8	78.9	77.3	64.5	65.3	54.3

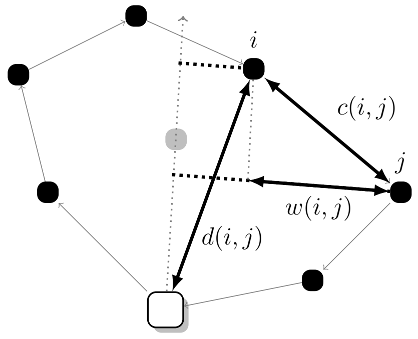


Figure 3: Metrics determining the 'badness' of edge (i, j). The *width* $w(i, j)$ is computed as the distance between nodes i and j measured along the axis perpendicular to the line connecting the depot and the route's center of gravity (gray dot). The *depth* $d(i, j)$ of this edge is the distance of the furthest node to the depot. The *cost* $c(i, j)$ is equal to the length of the edge.

in different routes that cross each other), the average length of edges connected to the depot, and the depth of routes (the distance between the depot and the furthest customer). Hereby, the prediction accuracy is higher, if the gap between the optimal and non-optimal solution is higher, or if the instances have more customers, as presented in Table 2.

In other words, moves that reduce the width of a route or increase its compactness are likely to make the solution better. For our heuristic, 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 (2017), 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.

Finally, the *depth* of an edge, i.e., the distance between the node that is furthest from the depot and the depot, has shown to improve the efficiency of the penalization. Even though the predictive power of the depth of routes was relatively low in the experiments, this metric provides additional information by describing the edge's relative position with respect to the depot. An explanation for why the penalization of deeper edges works well, is that the first two metrics width and cost tend to give more priority to edges that are closer to the depot (i.e., the first or last edges of a route). In other words, those edges tend to be wider, longer, or both in many instances. Consequently, the penalization of deep edges helps to diversify the search.

The three measures are illustrated in Fig. 3. With these three metrics *width*, *cost*, and *depth* we obtain an accurate spatial description of an edge, that can be readily transformed into a function that measures the 'badness' of an edge (i, j):

$$b(i,j) = \frac{[\lambda_w w(i,j) + \lambda_c c(i,j)] \left[\frac{d(i,j)}{\max_{(k,l)} d(k,l)} \right]^{\frac{\lambda_d}{2}}}{1 + p(i,j)}, \tag{3}$$

where $w(i,j)$ denotes the width of edge (i,j), $c(i,j)$ its cost, and $d(i,j)$ its depth. Since the distance of edges from the depot can vary significantly, we reduce the impact of the depth measure by means of a square root. Finally, we divide by the number of previously received penalties, so that the same edges are not penalized over and over. This is a standard approach in GLS (Voudouris and Tsang, 2003), and should enhance diversification of the search. Edge (i,j) with the highest value $b(i,j)$ is then penalized by incrementing $p(i,j)$. We also tried to embed the number of intersections per edge in the penalization function, since the number of intersection was a highly predictive feature in our analysis. However, including this metric did not result in significant improvements.

Depending on the respective instance or solution, we might also want to adapt our penalization function. For some instances it might be advantageous to penalize distant edges, whether for others we need to focus on expensive edges. We enable this adaptivity by defining binary variables λ_w, λ_c, and $\lambda_d \in \{0,1\}$ to switch features 'off' or 'on'. This results in 6 different feature combination and penalization functions (we neglect the options where $\lambda_c = \lambda_w = 0$). In pre-tests on the Golden instances, we determined how well each penalization function works, and observed that functions that include an edge's width ($\lambda_w = 1$) work slightly better than others. More formally, let the tuple $(\lambda_w, \lambda_c, \lambda_d)$ denote the penalization criterion, then we obtained the ranking $(1,0,0)$, $(1,1,0)$, $(1,0,1)$, $(1,1,1)$, $(0,1,0)$, $(0,1,1)$. We will use the penalization functions in this order in the heuristic, i.e., we will first penalize the widest edges, then the widest and longest edges, and so forth. A more detailed describtion follows in Section 4.

In summary, we have used findings from an exploratory study on VRP solution features to derive three characteristic features of edges. We use these edge features to define a penalization function, which can be adapted during the search. The penalization function detects bad edges that are then used as starting point for the local search.

4. A knowledge-driven local search heuristic

The local search operators in Section 2 can be readily combined with the knowledge-driven edge penalization in Section 3, to form a powerful yet simple heuristic. The heuristic starts by constructing an initial solution using the Clark-Wright heuristic (Clarke and Wright, 1964). It then identifies and penalizes the worst edge according to Eq. (3), and uses it as the starting point for the local search. The local search tries to remove the penalized edge by first using the EC operator, starting from the two adjacent nodes, and then the CE operator, starting from the penalized edge. The more penalizations an edge in the current solution has obtained, the more likely it will be removed by the local search. Whenever routes are changed by any of these two operators, the algorithm uses LK to re-optimize these routes. The local search leads to intensification, whereas the edge penalization results in diversification, as it allows the search to move to previously unexplored areas of the solution space. Since the local search is computationally inexpensive, these two steps (local search and edge penalization) can be iterated a large number of times.

Overall, the design is simple and can be readily extended, e.g., with more operators or functions.

4.1. Global optimization

One minor extension with a positive effect on performance is the optimization of the entire solution, i.e., the application of the local search operators to *all* edges and not only the worst one as determined by the penalization function. The rationale is that the standard local search that uses the penalization function emanates from specific edges, and therefore tends to focus on a specific part of the problem. Once a penalized edge has been successfully removed, the local search is terminated, even though the removal of the edge and the resulting change in the solution could trigger further changes in other parts of the solution which might yield further improvements. For instance, a successful EC operation could provide additional capacity in another route, which enables a CE operation, and so forth. Therefore, it seems reasonable to allow the local search to remove every possible edge, not only the worst edge, from time to time. In this global optimization step edge penalties are ignored. The heuristic first applies the EC operator, starting from all customer nodes, and then the CE operator, starting from all edges. Whenever a route is changed, it is reoptimized using the LK operator. This operation is repeated until no more improvements can be found.

This global optimization is computationally very expensive and we need to take care not to use it too often. We count the number of changes induced by EC and CE during 'normal' operation, i.e., when only focusing on penalized edges, and apply global optimization after a certain number of changes without improving the solution. We also apply global optimization whenever we find a new best solution to further intensify the search.

4.2. Restarts and penalization adaptation

Most GLS-based heuristics reset the penalties from time to time. The reason is that, as the search progresses, some edges might accumulate extremely high penalties, which might cause the heuristic to get stuck. Therefore, we reset the penalties of all edges when the heuristic does not find an improvement during a sufficiently large number of local changes. At the same time, we also reset the solution to the previously found global optimum, effectively restarting from the most promising solution. Restarts have shown to be one of the key drivers to find further improvements, especially during the later stages of the search.

These restart moments are also used to adapt the penalization function in the way described in Section 3, allowing the search to focus on another combination of edge attributes (width, cost, depth). The motivation is that, apparently, the previous edge penalization did not result in improvements for a large number of iterations, so that it seems appropriate to investigate an alternative way to determine the 'worst' edges. Before each restart, we try to apply all penalization functions, starting with the order that we presented in Section 3. We start with the most promising function (penalize wide edges), and switch to the next one when we did not find a better solution after a certain amount of local changes. After testing all penalization functions in this fashion without finding any improvement, we perform a restart, and change the order in which we test the functions, in such a way that the first penalization function to test becomes the last. Also, whenever we found a new global optimum, we move the current penalization function to the front, since it apparently worked well with the current solution. To make full use of the benefits of restarts, we test whether a restart would result in an improvement before changing the initial penalization function. Saving the current state (solution and penalties), we reset penalties and solution and apply each penalization function on this setup for a few iterations. If we do not find a new global best solution, we continue the search from the saved state.

In summary, the heuristic consists of only four components, three local search operators combined with edge penalization. The local search operators are complementary and focus on very specific areas of the solution, which makes them highly efficient. At each stage of the search, the computed solution is feasible with respect to all constraints. Since the edge selection as well as the local search is deterministic, the heuristic itself is completely deterministic and does not utilize any random components. The design of the heuristic is simple, and can be easily extended with regenerative or adaptive mechanisms. Finally, the local search components can also be replaced by simpler ones (e.g., replace EC with a simple swap and LK with 2-opt). This leads to a reduced implementation effort and a computational speed-up, at the cost of a reduced accuracy.

5. Analysis and verification of the heuristic

In the following, we aim to validate that the design of the described heuristic is (1) minimal in the sense that each component contributes to an improved solution quality, and (2) efficient in the sense that each component exhibits a significant positive effect on the performance of the heuristic that justifies its computational effort. For the heuristic to be both minimal and efficient, we should observe a better performance after the same computation time if all components are included, compared to a situation in which one or more components are left out.

We conduct the experiments on the 20 popular Golden instances (Golden et al., 1998). As the baseline, we compute solutions using the entire heuristic as described above with a maximal runtime of two minutes per instance. We compare the performance of this setup to the performance that we obtain when removing one of the local search components (CE, EC, LK). The results reveal the individual importance of each component for the functioning of the heuristic.

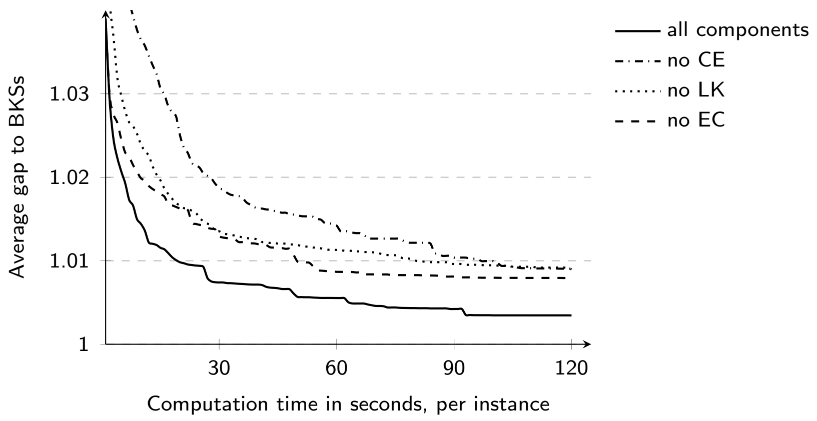


Figure 4: Average gap (in %) to the best known solutions (BKSs) over time on 20 instances by Golden et al. (1998) for different setups

From the results in Fig. 4 we can draw the following conclusions. Firstly, the heuristic performs best at any time, when all three local search components interact together. Thus, each component brings in a unique and efficient contribution to the heuristic orchestration. This is in line with our expectations, since we chose the three local search operators in such a way that each one focuses on a different optimization aspect (intra-route versus inter-route, two routes versus multiple routes). Secondly, all components seem to have a similar impact on the solution quality after two minutes of computations time, and the local search operators seems to have synergetic effects. Finally, the three local search components require a similar runtime per iteration across a variety of instances. For most instances CE and EC require the same amount of computation time whereas the time for LK is usually shorter, depending on the average size of routes. This equivalent runtime behaviour of components is beneficial for the heuristic design, since there is no ‘bottleneck’, i.e., a single operator that requires most of the computation time and thereby determines the overall runtime behaviour.

Similarly to above, we conduct a second set of experiments to investigate the importance of knowledge-driven edge penalization. So far, GLS implementations for the VRP have penalized the longest (or most expensive) edge. We compare this standard approach to our knowledge-driven approach, in which we use insights on properties of good solutions to describe an edge with more properties. Using the same instances as above, we observe performance improvements at any time with the knowledge-driven approach, as displayed in Fig. 5. Even though the differences are rather small, it is important to note that the standard approach is already highly effective and has led to one of the most efficient heuristics to date (Mester and Bräysy, 2007). Thus, any improvement is remarkable, especially since it requires only a slight change in the heuristic.

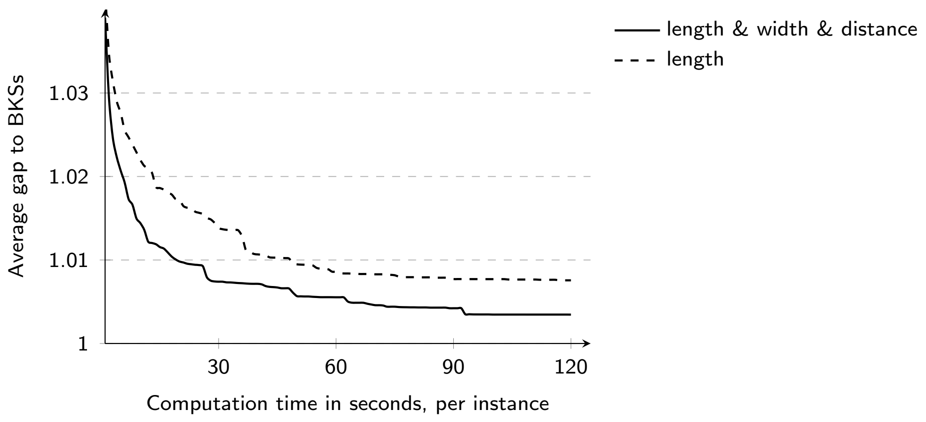


Figure 5: Average gap (in %) to the best known solutions (BKSs) over time on 20 instances by Golden et al. (1998) for different penalization functions

An interesting feature of the heuristic is that it is completely deterministic, i.e, it will always obtain the same solution after the same runtime for a certain instance. 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). The work by Hemmati and Hvattum (2017) is one of the few to shed light on the impact of randomness and whether it could be replaced with deterministic components. Even though randomness does trigger changes, these changes are mostly undirected, and in the extreme case one might argue that a good run on an instance is thus, to a certain degree, based on luck. As a consequence, it is more difficult to sufficiently test and analyze the heuristic, e.g., with experiments as we have performed them above, and generate a deeper understanding about why the heuristic works well. Not only are deterministic heuristics easier to analyze, it is also easier to evaluate their performance. Whereas stochastic heuristics might generate a different solution in every run and, therefore, need to be run multiple times on each instance, a deterministic heuristic only needs to be run a single time.

6. From VRP to other routing problems: an effective heuristic for the multi-depot VRP

Flexibility is a highly desirable property of vehicle routing heuristics (Cordeau et al., 2002). If a heuristic can tackle not only the standard VRP but a more generic class of routing problems with more constraints and attributes, we do not need to develop a new heuristic for each of the many variants (also called multi-attribute vehicle routing problems). In recent years some heuristics have shown that such generic solution approaches are attainable, most notably Vidal et al. (2013a).

In this section, we demonstrate that our simple heuristic design is suited to tackle a wider class of routing problems. Two features of the heuristic framework allow for its effortless adaptation to other routing variants. Firstly, the local search keeps solutions feasible at each stage of the search and never generates an infeasible solution. This implies that additional constraints only need to be validated within the local search components. If we impose a maximal length on the routes, for example, the heuristic only needs to check whether a local change would exceed the maximum route length (this requires only one extra line of source code per operator). Secondly, the edge penalization function presents an elegant tool to adapt the heuristic towards peculiarities of the respective routing problem. For example, whereas for the standard VRP it is preferable to penalize wide edges, for other problem variant we might want to focus on different features.

To demonstrate the versatility of our heuristic, we consider the Multi-Depot Vehicle Routing Problem. In the MDVRP customers can be delivered from a given set of depots. 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 without any major changes. 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 Clark-Wright heuristic. Since we have no additional constraints, we do not need to adapt the local search. Additionally, the edge penalization allows to tailor the search to the characteristics of the MDVRP. For instance, we found that the metric *depth* is more important to solve MDVRPs than to solve VRPs. The reason for this observation might be that it seems sensible to have edges that are relatively close to their assigned depots.

7. Computational Results

We test the performance of the heuristic on the basis of recent VRP benchmark sets, i.e., the prominent Golden instances ($\mathbb{G}$) (Golden et al., 1998), and the more recent Uchoa instances ($\mathbb{U}$) (Uchoa et al., 2017). While the $\mathbb{G}$ instances are based on regular geometric patterns like circles and squares, the $\mathbb{U}$ instances are more realistic and more diverse as they consider clustering of customers, variation in customer demand, and different depot locations, as shown in Fig. 6. Both sets include instances of relatively large size with up to 483 ($\mathbb{G}$) and 1001 ($\mathbb{U}$) customers, respectively. For the MDVRP we perform the experiments on the benchmark set by Cordeau ($\mathbb{C}$) (Cordeau et al., 1997).

We compare the performance of our heuristic to 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), the active guided evolution strategies (AGES) of Mester and Bräysy (2007), and the memetic algorithm (NB) of Nagata and Bräysy (2009). For the MDVRP instances, we compare our results with those of the HGSADC of Vidal et al. (2013b) and the adaptive large neighborhood search (ALNS) of Pisinger and Ropke (2007).

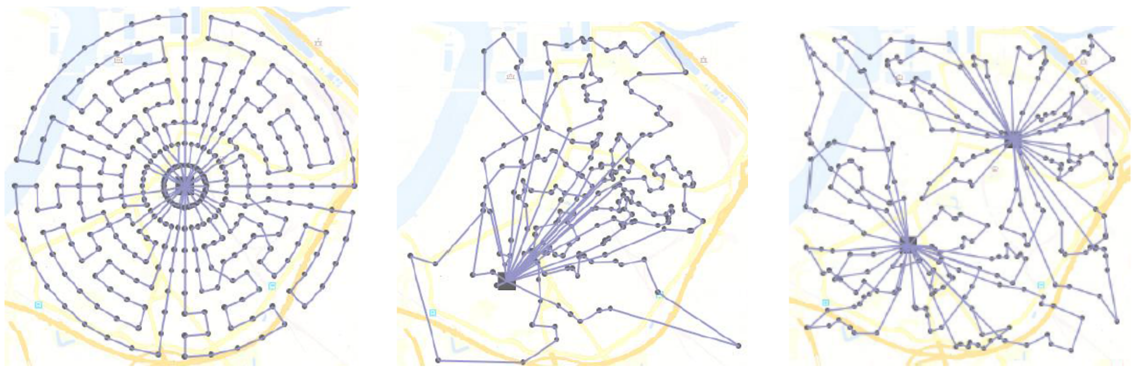


Figure 6: Benchmark instances from the $\mathbb{G}$-set, the $\mathbb{U}$-set and the $\mathbb{C}$-set

7.1. Parameter setup

The heuristic only has a few parameters that need to be tuned. We found that, in general, the precise choice of values only has a minor effect on the performance, even over a wide range of instances. In other words, the performance of the heuristic is relatively robust with respect to the chosen parameter values. Certain instances could be solved more accurately or more quickly with very specific parameter choices, but we chose to use one reasonable setup for almost all instances to demonstrate that the heuristic itself is capable of coping with different kinds of problems.

An EC operation is limited to three relocations at a time, since the benefits of higher values do not outweigh the increase in computation time. For each customer we consider the 30 nearest customers as possible neighbors in EC and CE operations. Even though on many instances this number is too high, we found it to be a good compromise between computational effort (which increases with more potential neighbors), and solution quality (which usually increases if more neighbors are considered). We increase the cost of penalized edges by 10%, as this value has been proven to work well in other heuristics (Kilby et al., 1999) and also worked well in our heuristic. Global optimization is applied after $N/10$ non-improving moves, where N is the number of customers in the instance. This value worked well in experimentation, even though different values yielded similar results. We found that lower values work better for MDVRP instances and, therefore, for those instances we set the value to $N/5$. The penalization function is changed after

$20N$ non-improving moves, and the solutions and penalties are reset after $100N$ changes without improvement, together with an adaptation of the penalization function. For most instances, this value is large enough to sufficiently explore the solution space with the current penalization function. For MDVRP instances and VRP instaces with constraints on route length, we found that the metric *depth* was more significant, and thus, we apply the penalization function $(1, 1, 1)$ first on those instances.

There are some $\mathbb{U}$-instances with an exceptionally large variance in customer demand (i.e., many customers with very small demand, and a few with a large demand, marked as SL in Uchoa et al. (2017)). This high variance results in visually chaotic routes with very long edges. For those instances, customers that are far away can also be promising neighbors, and, thus, we consider the 60 nearest customers for local search moves. To cope with the resulting increase in complexity, we limit EC operations to two relocations, and adapt and reset after $10N$ and $50N$, respectively. For a better scaling, we also choose this faster adaption and reset periods for instances with more than 500 customers. In summary, we observed as a general rule of thumb that the higher the variance in customer demand is, the more customer should be considered as potential neighbours, and the larger the instance, the more frequently one should adapt and reset.

We abort the search when the algorithm does not find any improvement for 3 minutes, and report the value and time of the best solution found. Due to the deterministic nature of the heuristic, multiple runs are not required. All experiments were executed on a Dell E5550 laptop with an Intel Core i7-5600 CPU working at 2.60GHz. Even though the computation times reported below are subject to this setup, the relative speed of an Intel 2.60GHz is compareable to the setup on which the other heuristics were tested, based on the analysis by Dongarra (2013) (e.g., the results reported for the $\mathbb{G}$-instances were normalized to the runtime of a Pentium IV 3.0 GHz in Vidal et al. (2013a)). The heuristic has been developed in Java, and was tested within the Eclipse development environment on Windows 7.

7.2. Results

The aggregated results on the VRP benchmark sets for our heuristic (denoted A&S) are presented in Table 3 and Table 4, detailed results per instance can be found in Appendix. All other heuristics include random components, and present their results as the average performance over a number of runs, whereas our heuristic produces the same solution in every run[2].

[2]An argument could be made that "the average cost over n runs" is not a reasonable way to represent the results of a stochastic optimization algorithm. Surely, many practitioners will not accept a claim like "we are 50% sure that the solution quality will not be worse than X", but rather demand more certainty. The solution occupying the 5th percentile, e.g., would make for a more useful claim "we are 95% sure that the solution quality will not be worse than X". This is akin to statistical hypothesis testing, where the null hypothesis is only rejected when $p < .5$, i.e., when we are 95% sure of our claim. One could therefore argue that typically, the way in which results are typically represented in the literature leads to an unfair bias towards the use of randomness.

Table 3: Results on the 20 $\mathbb{G}$ instances, as average over all instances, based on Vidal et al. (2013a)

AGES		NB		HGSADC		A&S	
Gap	Time	Gap	Time	Gap	Time	Gap	Time
0.327%	22.4	0.273%	29.2	0.267%	28.5	0.302%	2.2

Table 4: Results on the 100 $\mathbb{U}$ instances, as average over all instances, based on Uchoa et al. (2017)

	ILS		HGSADC		A&S	
N	Gap	Time	Gap	Time	Gap	Time
100-250	0.41%	3.2	0.11%	7.7	0.41%	1.8
250-500	0.75%	41.9	0.27%	45.2	0.65%	4.1
500-1000	0.97%	304.1	0.30%	318.2	0.71%	11.0

We observe that the heuristic computes competitive solutions for almost any instance. On the $\mathbb{G}$-benchmark-set it outperforms the best heuristics after short computation time on many instances. Similarly, the heuristic exhibits excellent performance on most $\mathbb{U}$ instances, and is on average better and faster than ILS and only about 0.35% worse than HGSADC.

It is especially successful on instances with many customers per route, since the local search puts a lot of emphasis on intra-route optimization and the exchanges of (possibly quite long) sequences. Reversely, the heuristic has more difficulties on instances with very short routes. On those instances, intra-route optimization and sequence exchanges are rarely possible, and most of the changes are triggered by EC. Likewise, some $\mathbb{U}$ instances with a very large variance in customer demand are difficult to solve, since they enforce the occurrence of extremely long edges. The results in Table 5 highlight that the heuristic performs as well as the best MDVRP heuristic in the literature and clearly outperforms the ALNS.

As can be seen from the results, our knowledge-driven heuristic computes high-quality solutions for almost any instance in a very short computing time. For the $\mathbb{G}$ instances, it competes with or even outperforms the state-of-the-art heuristics in less than 10% of the computation time, in some cases finding near-optimal solutions within a few seconds. This result is highlighted in Fig. 7. For a gap to the best-known solution

Table 5: Results on the 23 MDVRP $\mathbb{C}$ instances, as average over all instances, based on Vidal et al. (2013b)

ALNS		HGSADC		A&S	
Gap	Time	Gap	Time	Gap	Time
0.399%	3.8	0.056%	4.1	0.096%	0.5

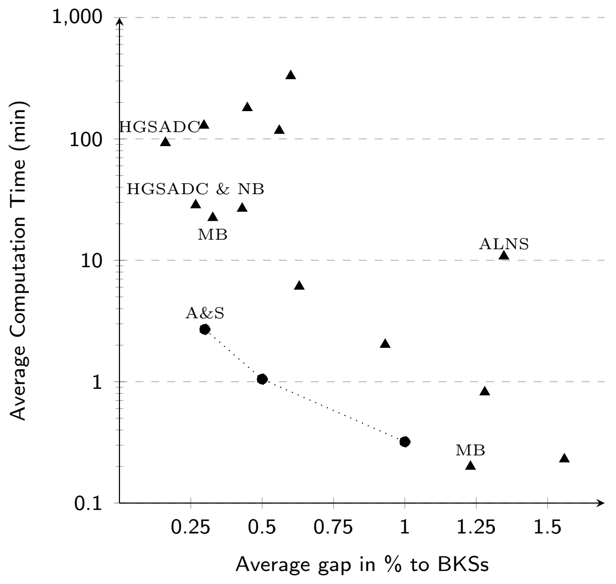


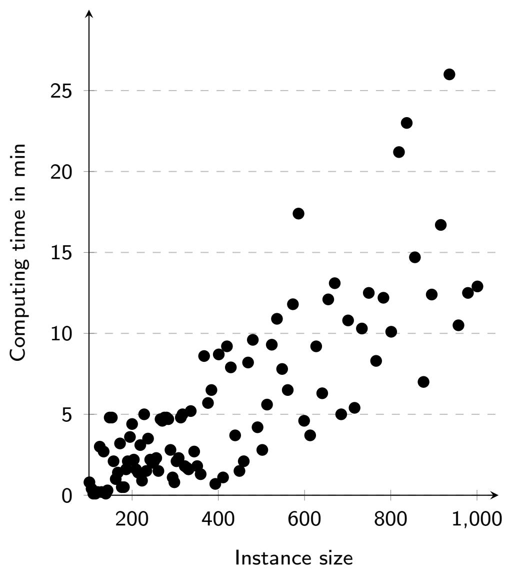
Figure 7: Performance versus computation time in comparison to state-of-the-art heuristics on $\mathbb{G}$ instances (based on Vidal et al. (2013a)).

of 0.30% or more, the heuristic sets the Pareto front in terms of quality–time trade-off. In other words, up to a very high quality level, our heuristic is the most-efficient heuristic in the literature for these instances.

On the $\mathbb{U}$ instances we observe similar results, especially for instances of larger size. For some instances, like P79, P90 and P98 the heuristic computes better results while being up to 40 times faster. In total, we outperform the other heuristics both, in solution quality and computation time, on 14 instances (in comparisson, ILS does so five times and HGSADC none, due to longer computation times). Likewise, the heuristic is almost 10 times faster on MDVRP instances compared to HGSADC, while showing minimal differences in terms of solution quality. In summary, it does not require more than a few minutes to compute solutions with very small gaps for VRP as well as MDVRP instances, resulting in a remarkable time-quality trade-off. This excellent time-quality trade-off opens a lot of potential to further improve the heuristic, and can be attributed to its scalability.

In Fig. 8 (left) we visualize the scaling of the heuristic's computing time with growing instance size. As can be seen, the heuristic computes solutions of similar quality, while the required computation time grows more or less as a linear function of the instance size. This 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.

All in all, the presented heuristic is one the most efficient heuristics in the routing literature. It consistently produces competitive solutions for a large variety of instances in a very short time. Additionally, its simple design enables straightforward extensions and adaptations to other routing problems.



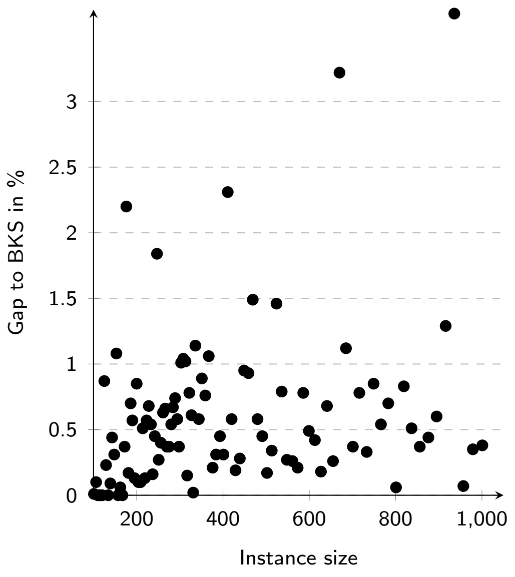


Figure 8: Computation time (left) versus performance (right) in dependence of instance size on the 100 $\mathbb{U}$ instances. While the computation time grows approximately linear in the instance size, the solution quality remains on a similar high level.

8. Conclusions and future research

In this paper we have presented a simple heuristic for the VRP that is entirely based on a highly efficient local search and guided through problem-specific knowledge. We implemented three complementary local search operators in such a way that they exhibit an almost linear runtime in most cases. 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 properties of VRP solutions. We demonstrated that all components contribute to the performance of the heuristic.

The resulting heuristic is simple in design, and yet is able to compete with the best heuristics in the literature on a wide range of benchmark instances. It computes high-quality solutions for large-scale instances in a few seconds or minutes, and is up to 40 times faster than state-of-the-art heuristics. 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 adding more local search operators, or by adapting the edge penalization through learning mechanisms. At the same time, we demonstrated that the heuristic can be readily used to solve other routing problems like the MDVRP.

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 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 definitely 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

Arnold, F. and Sörensen, K. (2017). What makes a solution good? the generation of problem-specific knowledge for heuristics. Working paper 3, University of Antwerp, Faculty of Applied Economics.

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.

Cordeau, J.-F., Gendreau, M., Laporte, G., Potvin, J.-Y., and Semet, F. (2002). A guide to vehicle routing heuristics. *Journal of the Operational Research society*, 53(5):512–522.

Dongarra, J. (2013). Performance of various computers using standard linear equations software. Technical report, Computer Science Department, University of Tennessee, Knoxville, Tennessee.

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.

Golden, B. L., Wasil, E. A., Kelly, J. P., and Chao, I.-M. (1998). The impact of metaheuristics on solving the vehicle routing problem: algorithms, problem sets, and computational results. In *Fleet management and logistics*, pages 33–56. Springer.

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.

Hemmati, A. and Hvattum, L. M. (2017). Evaluating the importance of randomization in adaptive large neighborhood search. *International Transactions in Operational Research*, 24(5):929–942.

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. (2005). Active guided evolution strategies for large-scale vehicle routing problems with time windows. *Computers & Operations Research*, 32(6):1593–1614.

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.

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

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

Prins, C. (2004). A simple and effective evolutionary algorithm for the vehicle routing problem. *Computers & Operations Research*, 31(12):1985–2002.

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

Rego, C. and Glover, F. (2007). Local search and metaheuristics. In *The traveling salesman problem and its variations*, pages 309–368. Springer.

Rousseau, L.-M., Gendreau, M., and Pesant, G. (2002). Using constraint-based operators to solve the vehicle routing problem with time windows. *Journal of heuristics*, 8(1):43–58.

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.

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 6: Results on the VRP $\mathbb{G}$ instances (Golden et al., 1998)

Instance	AGES			HGSADC			A&S		
	Value	Gap	Time	Value	Gap	Time	Value	Gap	Time
P01 (240)	5627.54	0.07	3.1	5627.00	0.06	11.7	5645.9	0.40	1.0
P02 (320)	8447.92	0.51	16.9	8446.65	0.50	20.8	8447.92	0.52	1.6
P03 (400)	11036.22	0	14.4	11036.22	0	28.0	11036.22	0	2.9
P04 (480)	13624.52	0.23	357.0	13624.52	0.23	43.7	13624.52	0.23	0.9
P05 (200)	6460.98	0	1.3	6460.98	0	2.6	6460.98	0	0.2
P06 (280)	8412.88	0.10	27.3	8412.90	0.10	8.4	8412.90	0.10	1.9
P07 (360)	10195.56	0.92	0.9	10157.63	0.54	22.9	10195.56	0.92	0.3
P08 (440)	11663.55	0.24	12.3	11646.58	0.10	40.7	11640.76	0.05	3.0
P09 (255)	583.39	0.63	6.0	581.79	0.36	16.2	581.9	0.38	1.5
P10 (323)	741.56	0.72	1.3	739.86	0.49	25.9	738.45	0.30	2.9
P11 (399)	918.45	0.61	7.4	916.44	0.39	45.6	915.89	0.33	3.9
P12 (483)	1107.19	0.41	10.8	1106.73	0.37	95.7	1106.42	0.34	6.6
P13 (252)	859.11	0.22	6.7	859.64	0.29	9.4	858.42	0.14	1.7
P14 (320)	1081.31	0.07	0.8	1082.41	0.17	14.1	1085.54	0.46	0.6
P15 (396)	1345.23	0.55	0.5	1343.52	0.42	39.2	1344.8	0.51	2.4
P16 (480)	1622.69	0.63	13.3	1621.02	0.53	58.3	1621.44	0.55	2.8
P17 (240)	707.79	0	0.5	708.09	0.05	7.1	707.76	0	3.9
P18 (300)	998.73	0.36	2.5	998.44	0.33	14.4	998.94	0.38	3.4
P19 (360)	1366.86	0.09	0.4	1367.83	0.16	27.9	1366.72	0.08	1.4
P20 (420)	1820.09	0.10	3.8	1822.02	0.20	38.2	1824.0	0.31	1.3
		0.327	22.4		0.267	28.5		0.302	2.2

Table 7: Results on the VRP 𝕌 instances (Uchoa et al., 2017)

Instance	ILS			HGSADC			A&S		
	Value	Gap	Time	Value	Gap	Time	Value	Gap	Time
P01 (101)	27591.0	0.00	0.1	27591.0	0.00	1.4	27595	0.01	0.8
P02 (106)	26375.9	0.05	2.0	26381.8	0.08	4.0	26388	0.10	0.4
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	0.1
P05 (120)	13337.6	0.04	1.7	13332.0	0.00	2.3	13332	0.00	0.2
P06 (125)	55673.8	0.24	1.4	55542.1	0.01	2.7	56024	0.87	3.0
P07 (129)	28998.0	0.20	1.9	28948.5	0.03	2.7	29006	0.23	0.2
P08 (134)	10947.4	0.29	2.1	10934.9	0.17	3.3	10916	0.00	2.7
P09 (139)	13603.1	0.10	1.6	13590.0	0.00	2.3	13602	0.09	0.1
P10 (143)	15745.2	0.29	1.6	15700.2	0.00	3.1	15769	0.44	0.3
P11 (148)	43452.1	0.01	0.8	43448.0	0.00	3.2	43583.0	0.31	4.8
P12 (153)	21400.0	0.85	0.5	21226.3	0.03	5.5	21450	1.08	4.8
P13 (157)	16876.0	0.00	0.8	16876.0	0.00	3.2	16876	0.00	2.1
P14 (162)	14160.1	0.16	0.5	14141.3	0.02	3.3	14147	0.06	1.0
P15 (167)	20608.7	0.25	0.9	20563.2	0.03	3.7	20557	0.00	1.4
P016 (172)	45616.1	0.02	0.6	45607.0	0.00	3.8	45779	0.37	3.2
P017 (176)	48249.8	0.92	1.1	47957.2	0.30	7.6	48864	2.20	0.5
P018 (181)	25571.5	0.01	1.6	25591.1	0.09	6.3	25612	0.17	0.5
P019 (186)	24186.0	0.17	1.7	24147.2	0.01	5.9	24313	0.70	1.6
P020 (190)	17143.1	0.96	2.1	16987.9	0.05	12.1	17077	0.57	2.1
P21 (195)	44234.3	0.02	0.9	44244.1	0.04	6.1	44284	0.13	3.6
P22 (200)	58697.2	0.20	7.5	58626.4	0.08	8.0	59076	0.85	4.4
P23 (204)	19625.2	0.31	1.1	19571.5	0.03	5.4	19584	0.10	2.2
P24 (209)	30765.4	0.36	3.8	30680.4	0.08	8.6	30686	0.10	1.6
P25 (214)	11126.9	0.25	2.3	10877.4	0.20	10.2	10911	0.51	1.4
P26 (219)	117595.0	0.00	0.9	117604.9	0.01	7.7	117743	0.13	3.1
P27 (223)	40533.5	0.24	8.5	40499.0	0.15	8.3	40666	0.57	0.9
P28 (228)	25795.8	0.21	2.4	25779.3	0.14	9.8	25918	0.68	5.0
P29 (233)	19336.7	0.55	3.0	19288.4	0.30	6.8	19333	0.54	1.5
P30 (237)	27078.8	0.14	3.5	27067.3	0.09	8.9	27086	0.16	3.5
P31 (242)	82874.2	0.15	17.8	82948.7	0.24	12.4	83121	0.45	2.2
P32 (247)	37507.2	0.63	2.1	37284.4	0.03	20.4	37960	1.84	2.0
		0.41	3.2		0.11	7.7		0.41	1.8

Table 8: Results on the VRP $\mathbb{U}$ instances (Uchoa et al., 2017)

Instance	ILS			HGSADC			A&S		
	Value	Gap	Time	Value	Gap	Time	Value	Gap	Time
P33 (251)	38840.0	0.40	10.8	38796.4	0.29	11.7	38788	0.27	2.1
P34 (256)	18883.9	0.02	2.0	18880.0	0.00	6.5	18956	0.40	2.3
P35 (261)	26869.0	1.17	6.7	26629.6	0.27	12.7	26724	0.63	1.5
P36 (266)	75563.3	0.11	10.0	75759.3	0.37	21.4	75974	0.66	4.7
P37 (270)	35363.4	0.21	9.1	35367.2	0.22	11.3	35420	0.37	4.6
P38 (275)	21256.0	0.05	3.6	21280.6	0.17	12.0	21323	0.37	4.8
P39 (280)	33769.4	0.80	9.6	33605.8	0.31	19.1	33683	0.54	4.8
P40 (284)	20448.5	1.10	8.6	20286.4	0.30	19.9	20361	0.67	4.7
P41 (289)	95450.6	0.28	16.1	95469.5	0.30	21.3	95886	0.74	2.8
P42 (294)	47254.7	0.19	12.4	47259.0	0.20	14.7	47441	0.58	1.1
P43 (298)	34356.0	0.37	6.9	34292.1	0.18	10.9	34357	0.37	0.8
P44 (303)	21895.8	0.69	14.2	21850.9	0.49	17.3	21963	1.01	2.1
P45 (308)	26101.1	0.94	9.5	25895.4	0.14	15.3	26127	1.04	2.3
P46 (313)	94297.3	0.27	17.5	94265.2	0.24	22.4	94999	1.02	4.8
P47 (317)	78356.0	0.00	8.6	78387.8	0.04	22.4	78473	0.15	5.0
P48 (322)	29991.3	0.42	14.7	29956.1	0.30	15.2	30100	0.78	1.8
P49 (327)	27812.4	0.93	19.1	27628.2	0.26	18.2	27724	0.61	1.7
P50 (331)	31235.5	0.43	15.7	31159.6	0.18	24.4	31109	0.02	1.6
P51 (336)	139461.0	0.19	21.4	139534.9	0.24	38.0	140789	1.14	5.2
P52 (344)	42284.0	0.44	22.6	42208.8	0.26	21.7	42343	0.58	2.7
P53 (351)	26150.3	0.79	25.2	26014.0	0.26	33.7	26176	0.89	1.8
P54 (359)	52076.5	1.10	48.9	51721.7	0.41	34.9	51898	0.76	1.3
P55 (367)	23003.2	0.83	13.1	22838.4	0.11	22.0	23056	1.06	8.6
P56 (376)	147713.0	0.00	7.1	147750.2	0.03	28.3	148021	0.21	5.7
P57 (384)	66372.5	0.44	34.5	66270.2	0.29	40.2	66284	0.31	6.5
P58 (393)	38457.4	0.49	20.8	38374.9	0.28	28.7	38443	0.45	0.7
P59 (401)	66715.1	0.71	60.4	66365.4	0.18	49.5	66448	0.31	8.7
P60 (411)	19954.9	1.20	23.8	19743.8	0.13	34.7	20173	2.31	1.1
P61 (420)	107838.0	0.04	22.2	107924.1	0.12	53.2	108428	0.58	9.2
P62 (429)	65746.6	0.37	38.2	65648.5	0.23	41.5	65624	0.19	7.9
P63 (439)	36441.6	0.13	39.6	36451.1	0.15	34.6	36497	0.28	3.7
P64 (449)	56204.9	1.53	59.9	55553.1	0.35	64.9	55881	0.95	1.5
P65 (459)	24462.4	1.16	60.6	24272.6	0.38	42.8	24407	0.93	2.1
P66 (469)	222182.0	0.12	36.3	222617.1	0.32	86.7	225208	1.49	8.2
P67 (480)	89871.2	0.38	50.4	89760.1	0.25	67.0	90057	0.58	9.6
P68 (491)	67226.7	0.89	52.2	66898.0	0.40	71.9	66934	0.45	4.2
		0.75	41.9		0.27	45.2		0.65	4.1

Table 9: Results on the VRP $\mathbb{U}$ instances (Uchoa et al., 2017)

Instance	ILS			HGSADC			A&S		
	Value	Gap	Time	Value	Gap	Time	Value	Gap	Time
P69 (502)	69346.8	0.14	80.8	69328.8	0.11	63.6	69373	0.17	2.8
P70 (513)	24434.0	0.96	35.0	24296.6	0.40	33.1	24284	0.34	5.6
P71 (524)	155005.0	0.27	27.3	154979.5	0.25	80.7	156853	1.46	9.3
P72 (536)	95700.7	0.61	62.1	95330.6	0.22	107.5	95870	0.79	10.9
P73 (548)	86874.1	0.19	64.0	86998.5	0.33	84.2	86944	0.27	7.8
P74 (561)	43131.3	0.88	68.9	42866.4	0.26	60.6	42868	0.26	6.5
P75 (573)	51173.0	0.77	112.0	50915.1	0.27	188.2	50888	0.21	11.8
P76 (586)	190919.0	0.20	78.5	190838.0	0.15	175.3	192035	0.78	17.4
P77 (599)	109384.0	0.52	73.0	109064.2	0.23	125.9	109346	0.49	4.6
P78 (613)	60444.2	1.11	74.8	59960.0	0.30	117.3	60031	0.42	3.7
P79 (627)	62905.6	0.87	162.7	62524.1	0.25	239.7	62477	0.18	9.2
P80 (641)	64606.1	1.20	140.4	64192.0	0.55	158.8	64273	0.68	6.3
P81 (655)	106782.0	0.00	47.2	106899.1	0.11	150.5	107058	0.26	12.1
P82 (670)	147676.0	0.66	61.2	147222.7	0.35	264.1	151424	3.22	13.1
P83 (685)	68988.2	0.82	73.9	68654.1	0.33	156.7	69189	1.12	5.0
P84 (701)	83042.2	0.91	210.1	82487.4	0.24	253.2	82597	0.37	10.8
P85 (716)	44171.6	1.49	225.8	43641.4	0.27	264.3	43866	0.78	5.4
P86 (733)	137045.0	0.50	111.6	136587.6	0.16	244.5	136818	0.33	10.3
P87 (749)	78275.9	0.74	127.2	77864.9	0.21	313.9	78362	0.85	12.5
P88 (766)	115738.0	0.92	242.1	115147.9	0.41	383.0	115301	0.54	8.3
P89 (783)	73722.9	1.37	235.5	73009.6	0.39	269.7	73234	0.70	12.2
P90 (801)	74005.7	0.57	432.6	73731.0	0.20	289.2	73634	0.06	10.1
P91 (819)	159425.0	0.51	148.9	158899.3	0.18	374.3	159921	0.83	21.2
P92 (837)	195027.0	0.39	173.2	194476.5	0.11	463.4	195262	0.51	23.0
P93 (856)	89277.6	0.24	153.7	89238.7	0.20	288.4	89391	0.37	14.7
P94 (876)	100417.0	0.70	409.3	99884.1	0.17	495.4	100155	0.44	7.0
P95 (895)	54958.5	1.45	410.2	54439.8	0.49	321.9	54499	0.60	12.4
P96 (916)	330948.0	0.33	226.1	330198.3	0.11	560.8	334077	1.29	16.7
P97 (936)	134530.0	1.07	202.5	133512.9	0.31	531.5	137994	3.67	26.0
P98 (957)	85936.6	0.31	311.2	85822.6	0.18	432.9	85735	0.07	10.5
P99 (979)	120253.0	0.89	687.2	119502.1	0.26	554.0	119614	0.35	12.5
P100 (1001)	73985.4	1.71	792.8	72956.0	0.29	549.0	73016	0.38	12.9
		0.97	304.1		0.30	318.2		0.71	11.0

Table 10: Results on the MDVRP $\mathbb{C}$ instances (Cordeau et al., 1997)

Instance	D	ALNS			HGSADC			A&S		
		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	473.86	0.00	0.1
P03 (75)	2	641.19	0.00	1.1	641.19	0.00	0.4	641.19	0.00	0.1
P04 (100)	2	1006.9	0.49	1.5	1001.23	0.00	1.9	1002.59	0.14	0.7
P05 (100)	2	752.3	0.31	2	750.03	0.00	1.1	750.03	0.00	0.2
P06 (100)	3	883.01	0.74	1.6	876.50	0.00	1.1	876.70	0.02	0.2
P07 (100)	4	889.36	0.84	1.5	884.43	0.28	1.6	881.97	0.00	0.7
P08 (249)	2	4421.03	1.10	5.6	4397.42	0.56	10.0	4393.13	0.47	0.7
P09 (249)	3	3892.50	0.88	6.0	3868.59	0.26	9.5	3868.59	0.26	2.9
P10 (249)	4	3666.85	0.98	6.1	3636.08	0.14	9.8	3651.25	0.55	0.5
P11 (249)	4	3573.23	0.77	6.0	3548.25	0.06	7.1	3558.09	0.34	0.4
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 (5160)	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	2709.9	0.00	0.4
P18 (240)	6	3736.53	0.91	7.0	3702.85	0.00	4.5	3702.85	0.00	1.2
P19 (240)	6	3838.76	0.31	5.3	3827.06	0.00	4.2	3828.61	0.04	0.4
P20 (240)	6	4064.76	0.16	5.0	4058.07	0.00	4.4	4058.07	0.00	0.5
P21 (360)	6	5501.58	0.46	9.7	5476.41	0.00	10.0	5494.19	0.32	0.3
P22 (360)	6	5722.19	0.35	7.7	5702.16	0.00	10.0	5706.81	0.08	0.2
P23 (360)	6	6092.66	0.23	7.4	6078.75	0.00	10.0	6078.75	0.00	0.6
			0.399	3.8		0.056	4.1		0.096	0.5