Efficient local search strategies for the Mixed Capacitated Arc Routing Problems under Time restrictions with Intermediate Facilities

Elias J. Willemse[a,1,*], Johan W. Joubert[a,2]

[a]*Centre for Transport Development, Department of Industrial and Systems Engineering, University of Pretoria, South Africa, 0002*

Abstract

In this paper, we extend our previous work on greedy constructive heuristics for the Mixed Capacitated Arc Routing Problem under Time Restrictions with Intermediate Facilities (MCARPTIF) by developing efficient Local Search improvement heuristics for the problem. Five commonly used arc routing move operators were adapted for the problem, and basic Local Search implementations were tested on waste collection benchmark sets. Tests showed that despite the application of commonly used speed-up techniques, the Local Search implementations are very slow on large test instances with more than one-thousand required arcs and edges. In response, more advanced Local Search acceleration mechanisms from literature were adapted and combined for the MCARPTIF and tested on the same instances. On the large instances, the basic Local Search setups took between fifteen minutes and three hours to improve a single solution to local optima, whereas the accelerated implementations took at most four minutes while producing similar quality local optima. The best performing implementation made use of two existing acceleration mechanisms, namely Static-Move-Descriptors and Greedy-Compound-Independent-Moves. A third mechanism, Nearest-Neighbour-Lists was also tested and although it reduced execution times it resulted in Local Search terminating at worse quality local optima. Given the importance of Local Search within metaheuristics, the developed Local Search heuristics provide a significant contribution to arc routing. First, the implementations can be extended to and incorporated into metaheuristics for the MCARPTIF. Second, the acceleration mechanisms can be applied to existing Local Search metaheuristics for Capacitated Arc Routing Problems, thereby improving the efficiency of the metaheuristics and allowing them to better deal with large instances.

Keywords: Waste management, Capacitated Arc Routing Problem, Mixed network, Intermediate Facilities, Time restrictions, Local Search

1. Introduction

In this paper, we extend our previous work in [25, 27] on constructive heuristics for the Mixed Capacitated Arc Routing Problem under Time Restrictions with Intermediate Facilities (MCARPTIF) by developing efficient Local Search based improvement heuristics for the problem. The MCARPTIF, which closely models residential waste collection, is a generalisation of the classical Capacitated Arc Routing Problem (CARP), first proposed by Golden and Wong [12]. The objective of the MCARPTIF is to determine routes of minimal total cost for a fleet of homogeneous vehicles so that each road segment with demand, representing waste to be collected, is serviced exactly once by a vehicle. The fleet size can be either unlimited, which is the version dealt with in this paper, limited, or left as a decision variable and total cost can also be measured in total distance travelled or the sum of the time taken to complete all routes. Consistent with the Mixed Capacitated Arc Routing Problem (MCARP), studied in [1, 7, 13, 15], the MCARPTIF takes into consideration a mixed road network that can consist of one-way streets that can be traversed or serviced in only one direction, busy two-way streets that require each side to be serviced separately, and two-way

*Corresponding author

Email addresses: ejwillemse@gmail.com (Elias J. Willemse), johan.joubert@up.ac.za (Johan W. Joubert)

[1]Tel: +27 12 420 3443; Fax: +27 12 362 5103 (E.J. Willemse)

[2]Tel: +27 12 420 2843; Fax: +27 12 362 5103 (J.W. Joubert)

streets that can be traversed or serviced in either direction. Consistent with the Capacitated Arc Routing Problem under Time Restrictions with Intermediate Facilities (CARPTIF), first proposed in [11], the MCARPTIF also accounts for Intermediate Facilities (IFs), representing waste transfer stations and landfills, where vehicles are allowed to unload their waste and resume their collection rounds.

The sum of demand collected on a subtrip between IF visits may then not exceed vehicle capacity and the route must include a final IF visit before returning to the depot. Furthermore, the total time of a vehicle route may not exceed a time restriction, typically equal to the available working hours per shift. For the latest review of Arc Routing Problems, including the CARP, MCARPTIF and MCARP, we refer the reader to Corberán and Laporte [5] and Mourão and Pinto [17], which builds on the previous reviews of Dror [9] and Corberán and Prins [6].

Since the CARP and all its extensions are $\mathscr{NP}$-hard the most effective methods for solving the problems are based on heuristic and metaheuristic methods [6]. In [25, 27], we developed constructive heuristics to generate initial solutions for the MCARPTIF, but computational tests showed that their performance is inconsistent, and the generated initial solutions leave room for improvement. When higher quality solutions are required and when sufficient computing time is available, metaheuristics can be used to generate improved solutions. For a general review of metaheuristics, we refer the reader to [21]. Despite their popularity, Prins [19] state that CARP metaheuristics currently have certain limitations, one of which is that they cannot yet produce solutions within reasonable computing times for large instances met in real applications, such as waste collection. Furthermore, no metaheuristic implementations are currently available for the MCARPTIF. As a first step to address the two research gaps this paper presents efficient Local Search (LS) improvement heuristics that are capable of dealing with large MCARPTIF instances. The motivation for focussing on LS is that it is a widely applied improvement method and the core optimisation component of metaheuristic strategies for CARPs. In fact, all recent CARP metaheuristics reviewed by Muyldermans and Pang [18] and Prins [19] rely on some form of LS.

Studies on LS with the aim of improving its efficiency are not uncommon, and include, amongst others, the work of Beullens et al. [2], Chen and Hao [3] and Vidal [22] who focus on CARPs, and the work of Ergun et al. [10], Irnich et al. [14] and [28] who focus on the VRP, which is the node routing equivalent of the CARP. Except for [14], the advantages of more efficient LS implementations are ultimately demonstrated using LS-based metaheuristics. However, all these studies focussed on existing problem variants with their underlying LS components, such as move operators, well established and documented. Although it would be extremely valuable for the MCARPTIF, we consider such a full study, focussing on both the efficacy of LS and the impact thereof on to-be-developed MCARPTIF metaheuristics, to be beyond the scope of one paper. The focus of this paper is therefore exclusively on efficient LS heuristics for the problem. For preliminary results for the full study, including tests on the MCARP, we refer the reader to [23].

To develop efficient LS heuristics, we first adapted existing LS components, developed in [1, 15] for the MCARP, to the MCARPTIF. Then, to improve the efficiency of our implementations on large problem instances, we adapted and combined three existing LS acceleration methods, namely Greedily Compounding Independent Moves [10], Static Move Descriptors [28], and Nearest Neighbour Lists [2]. To our knowledge, this paper presents the first results on their combined usage. We tested the accelerated LS setups on MCARPTIF instances available from [26], and results showed that the three methods significantly improve the efficiency of LS on large instances, but most often at a trade-off in solution quality. As a final test, we directly compared all our LS implementations using their average execution times to reach local optima and percentage cost improvement over initial solutions. The analysis was then used to identify non-dominated implementations, and to make a final recommendation on to the LS setup that can be extended to and incorporated into metaheuristic solution methods for the MCAPRTIF.

The rest of the paper is structured as follows. In the next section we formally introduce MCARPTIF terms, symbols and its solution representation that are used throughout the paper. This is followed in Section 3 by a brief review of LS and its commonly used moved operators for the MCARP and existing LS acceleration mechanisms that have been developed for the CARP and VRP. In Section 4 we show how the MCARP move operators can be adapted to the MCARPTIF. Section 5 is dedicated to our adaptation and combination of existing LS acceleration mechanisms to the MCARPTIF. Computational results are presented in Section 6 followed by a discussion of our main findings and opportunities for further research.

2. Problem notation and solution representation

To better explain and review existing LS implementations for CARPs, we first introduce basic MCARPTIF terms and symbols and its solution representation. Consistent with the notation of Willemse and Joubert [25], the MCARPTIF considers a graph $\boldsymbol{G} = (\boldsymbol{V}, \boldsymbol{E} \cup \boldsymbol{A})$, where $\boldsymbol{V}$ represents the set of vertices, $\boldsymbol{E}$ represents the set of undirected edges where an edge links two vertices and may be traversed in both directions, and $\boldsymbol{A}$ represents the set of arcs where an arc also links two vertices but can only be traversed in one direction. Edges and arcs are given in the form $(v_i, v_j) \in \boldsymbol{E}$ where $v_i, v_j \in \boldsymbol{V}$ represent the start and end vertices of the arc or edge. For waste collection, $\boldsymbol{V}$ corresponds to road intersections and dead-ends, while $\boldsymbol{E}$ and $\boldsymbol{A}$ model road segments between vertices. A subset of required edges and arcs, $\boldsymbol{E}_r \subseteq \boldsymbol{E}$ and $\boldsymbol{A}_r \subseteq \boldsymbol{A}$, must be serviced by a fleet of K homogeneous vehicles with limited capacity, Q, that are based at the depot vertex, v_0. The fleet size K can be either fixed, left as a decision variable or treated as unlimited. Vehicles are allowed to unload their waste at any Intermediate Facility (IF) at the cost of λ and resume their collection routes. At the end of its route, a vehicle must first visit an IF before returning to the depot. The set of IFs is modelled in $\boldsymbol{G}$ as $\boldsymbol{\Gamma}$, where $\boldsymbol{\Gamma} \subset \boldsymbol{V}$. The sum of demand on each sub-trip between IF visits may not exceed Q, and unless $v_0 \in \boldsymbol{\Gamma}$, a vehicle has to visit an IF before returning to the depot. Lastly, a route length or time restriction of L is imposed on each vehicle route, typically corresponding to available work hours in a day.

To solve the MCARPTIF, the graph $\boldsymbol{G}$ is transformed into a fully directed graph, $\boldsymbol{G}^* = (\boldsymbol{V}, \boldsymbol{A}^*)$, by including all arcs $\boldsymbol{A}$ in $\boldsymbol{A}^*$, and by replacing each edge, $(v_i, v_j) \in \boldsymbol{E}$, with two opposite arcs, $\{(v_i, v_j), (v_j, v_i)\} \in \boldsymbol{A}^*$. Arcs in $\boldsymbol{A}^*$ are identified by indices from 1 to β, where $\beta = |\boldsymbol{A}^*|$. Each arc $u^* \in \boldsymbol{A}^*$ has a deadheading time, $c(u^*)$, denoting the time of traversing the arc without servicing it. The cost of the shortest path between arcs u^* and $v^* \in \boldsymbol{A}^*$, which excludes the costs of deadheading u^* and v^*, is given by $D(u^*, v^*)$, which is pre-calculated for all arcs in $\boldsymbol{A}^*$. Shortest paths can be efficiently calculated using a modified version of Dijkstra's algorithm, and may also incorporate forbidden turns and turn-penalties [15]. The *Floyd-Warshall* shortest-path algorithm can also be adapted. Details for this can be found in [23]. The depot is modelled by including in $\boldsymbol{A}^*$ a fictitious loop, $\sigma = \{v_0, v_0\}$, with zero deadheading and service times. Similarly, the set of IFs are modelled in $\boldsymbol{A}^*$ as a set of dummy arcs, $\boldsymbol{I}$, such that each IF in $\boldsymbol{\Gamma}$ is modelled as a fictitious loop, $\Phi_i \in \boldsymbol{I}$, and Φ_i also has zero deadheading times.

Required arcs, $\boldsymbol{A}_r$, and edges, $\boldsymbol{E}_r$, of $\boldsymbol{G}$ correspond in $\boldsymbol{G}^*$ to a subset $\boldsymbol{R} \subseteq \boldsymbol{A}^*$ of required arcs. Each required arc, $u \in \boldsymbol{R}$, has a demand, $q(u)$, a servicing time, $w(u)$, and a pointer, $inv(u)$, to the arc between the same vertices but in the opposite direction. Each required arc in the original graph, $\boldsymbol{G}$, is coded in $\boldsymbol{R}$ by one arc, u, with $inv(u) = 0$, while each required edge is encoded as two opposite arcs, u and v, such that $inv(u) = v$ and $inv(v) = u$. Where u represents the edge $(v_i, v_j) \in \boldsymbol{E}$ in the original graph, $inv(u)$ will represent (v_j, v_i). The best IF to visit after servicing arc u and before servicing arc v can be pre-calculated using

$$\Phi^*(u, v) = \arg\min\{D(u, k) + D(k, v) : k \in \boldsymbol{I}\}, \tag{1}$$

$$\mu^*(u, v) = D(u, \Phi^*(u, v)) + D(\Phi^*(u, v), v) + \lambda, \tag{2}$$

where $\Phi^*(u, v)$ gives the best IF to visit, and $\mu^*(u, v)$ gives the duration of the visit, including the unloading time, λ, and deadheading time.

An MCARPTIF solution, $\boldsymbol{T}$, is a list, $[\boldsymbol{T}_1, \ldots, \boldsymbol{T}_{|\boldsymbol{T}|}]$, of $|\boldsymbol{T}|$ vehicle routes. Each route, $\boldsymbol{T}_i$, is a list of subtrips $[\boldsymbol{T}_{i,1}, \ldots, \boldsymbol{T}_{i,|\boldsymbol{T}_i|}]$, and each subtrip, $\boldsymbol{T}_{i,j}$, consists of a sequence, $[T_{i,j,1}, \ldots, T_{i,j,|\boldsymbol{T}_{i,j}|}]$, of required arcs and edges to be serviced, referred to simply as tasks. It is assumed that the shortest path is always followed between consecutive tasks $T_{i,j,k}$ and $T_{i,j,k+1}$. Thereby, the deadheading of arcs and edges between $T_{i,j,k}$ and $T_{i,j,k+1}$ is not explicitly included in $\boldsymbol{T}$, but their deadheading costs is still accounted for via $D(T_{i,j,k}, T_{i,j,k+1})$. Further, the first subtrip, $\boldsymbol{T}_{i,1}$, in a route always starts at the depot. All other subtrips, excluding the last one, starts and ends with the best IF visits, given by Equation (1). The last subtrip ends with an IF and depot visit. With their direct inclusion in $\boldsymbol{T}$, the depot and dummy arcs are assigned servicing times and demand of zero.

The capacity of each subtrip is calculated as $\sum_{n=1}^{|\boldsymbol{T}_{i,j}|} w(T_{i,j,n})$, which may not exceed Q. The durations of a subtrip

and a route, and the cost of a solution is calculated using Equations (3) to (5):

$$Z_{subtrip}(\boldsymbol{T}_{i,j}) = \sum_{n=1}^{|\boldsymbol{T}_{i,j}|-1} \Big(D(T_{i,j,n}, T_{i,j,n+1})\Big) + \sum_{n=1}^{|\boldsymbol{T}_{i,j}|} w(T_{i,j,n}) + \lambda \tag{3}$$

$$Z_{route}(\boldsymbol{T}_i) = \sum_{j=1}^{|\boldsymbol{T}_i|} Z_{subtrip}(\boldsymbol{T}_{i,j}), \tag{4}$$

$$Z(\boldsymbol{T}) = \sum_{i=1}^{|\boldsymbol{T}|} Z_{route}(\boldsymbol{T}_i), \tag{5}$$

where $Z_{route}(\boldsymbol{T}_i) \leq L$.

In the subsequent review, when referring to an MCARP solution, also denoted by $\boldsymbol{T}$, it consists of a list, $[\boldsymbol{T}_1, \ldots, \boldsymbol{T}_K]$, of K vehicle routes, and each route, $\boldsymbol{T}_i \in \boldsymbol{T}$, consists of a list of tasks $[T_{i,1}, \ldots, T_{i,|\boldsymbol{T}_i|}]$ with $T_{i,1} = T_{i,|\boldsymbol{T}_i|} = \sigma$. Its cost, also denoted $Z(\boldsymbol{T})$, is similarly calculated as with the MCARPTIF.

3. A review of Local Search for CARPs and acceleration mechanisms

3.1. Local Search for the MCARP

In its basic form, LS starts with an initial solution and iteratively moves to an improving solution belonging to the neighbourhood of the current one. Let $\boldsymbol{T} \in \boldsymbol{X}$ be a feasible solution for the MCARP where $\boldsymbol{X}$ is the set of all feasible solutions, and let $Z(\boldsymbol{T})$ be the cost of the solution. The neighbourhood, $\mathcal{N}$, is a mapping $\mathcal{N} : \boldsymbol{X} \rightarrow 2^{\boldsymbol{X}}$, and each element $\boldsymbol{T}' \in \mathcal{N}(\boldsymbol{T})$ is called a neighbour of $\boldsymbol{T}$. Neighbours with cost $Z(\boldsymbol{T}') < Z(\boldsymbol{T})$ are improving neighbours. LS starts with a given initial solution $\boldsymbol{T}^{(0)} \in \boldsymbol{X}$. In each iteration t, local search replaces the current solution $\boldsymbol{T}^{(t)}$ by an improving neighbour $\boldsymbol{T}^{(t+1)} \in \mathcal{N}(\boldsymbol{T}^{(t)})$. The search terminates when a local optimum is reached, meaning there are no improving neighbours in $\mathcal{N}(\boldsymbol{T}^{(t)})$. For a comprehensive review of LS, we refer the reader to [16]. Typically, the neighbourhood is defined implicitly by a set of moves $\mathscr{M}$, where each move, $\pi \in \mathscr{M}$, transforms the current solution into a neighbouring one. For CARPs, moves typically change the position of tasks between and within routes.

Belenguer et al. [1] and Lacomme et al. [15] use seven move operators within their LS implementations for the MCARP, which they embed within Memetic Algorithms. As noted by Prins [19], such move operators have been extensively used within other CARP metaheuristics, including the recent implementations of Chen and Hao [3], Chen et al. [4] and Vidal [22], and which we also adapted to the MCARPTIF. Five of the seven MCARP move operators are shown in Figure 1. For illustrative purposes, a_i represents an arc task with $inv(a_i) = 0$ and e_i represents an edge task with $inv(e_i) = e'_i$. An edge task can be serviced in either of its directions, e_i or e'_i. Lastly, let u and v be two different tasks, which can be in the same or different routes.

The first operator, *relocate*, moves task u before task v, which can be in the same or a different route. It also considers the special case to insert u after v if v is the last task of its route. The second operator, *double-relocate*, is similar to *relocate*, therefore not illustrated in Figure 1, with adjacent tasks moved together to a new position. The *exchange* operator, shown in the figure, exchanges the positions of two tasks, u and v, and the third operator, *flip*, inverts an edge task u so that the edge is traversed in its opposing direction. Advanced versions of the operator are used by Beullens et al. [2] and Vidal [22] in which the optimal orientation of all tasks in a route is efficiently determined.

The rest of the MCARP operators employ *two-opt* moves that first delete links in the visitation sequence and then relinks the route segments to create new routes. In certain cases, the route segments are first reversed before being relinked. Three relinking options are considered, each constituting a different move. The first move, *two-opt-1*, is applied when u and v are in the same route, and *two-opt-2* and *two-opt-3* are applied when they are in different routes. All three *two-opt* moves are illustrated in Figure 1. *Two-opt-2* is more intuitively referred to as *cross* by Beullens et al. [2] since the move results in end portions of the routes being crossed. It is also easier to implement compared to the other *two-opt* moves since it does not result in segments being reversed.

Two-opt-1 and *two-opt-3* involve the reversal of certain route segments to be relinked. For the symmetric CARP, this can be automatically done without any additional calculations. The same does not always hold for the asymmetric

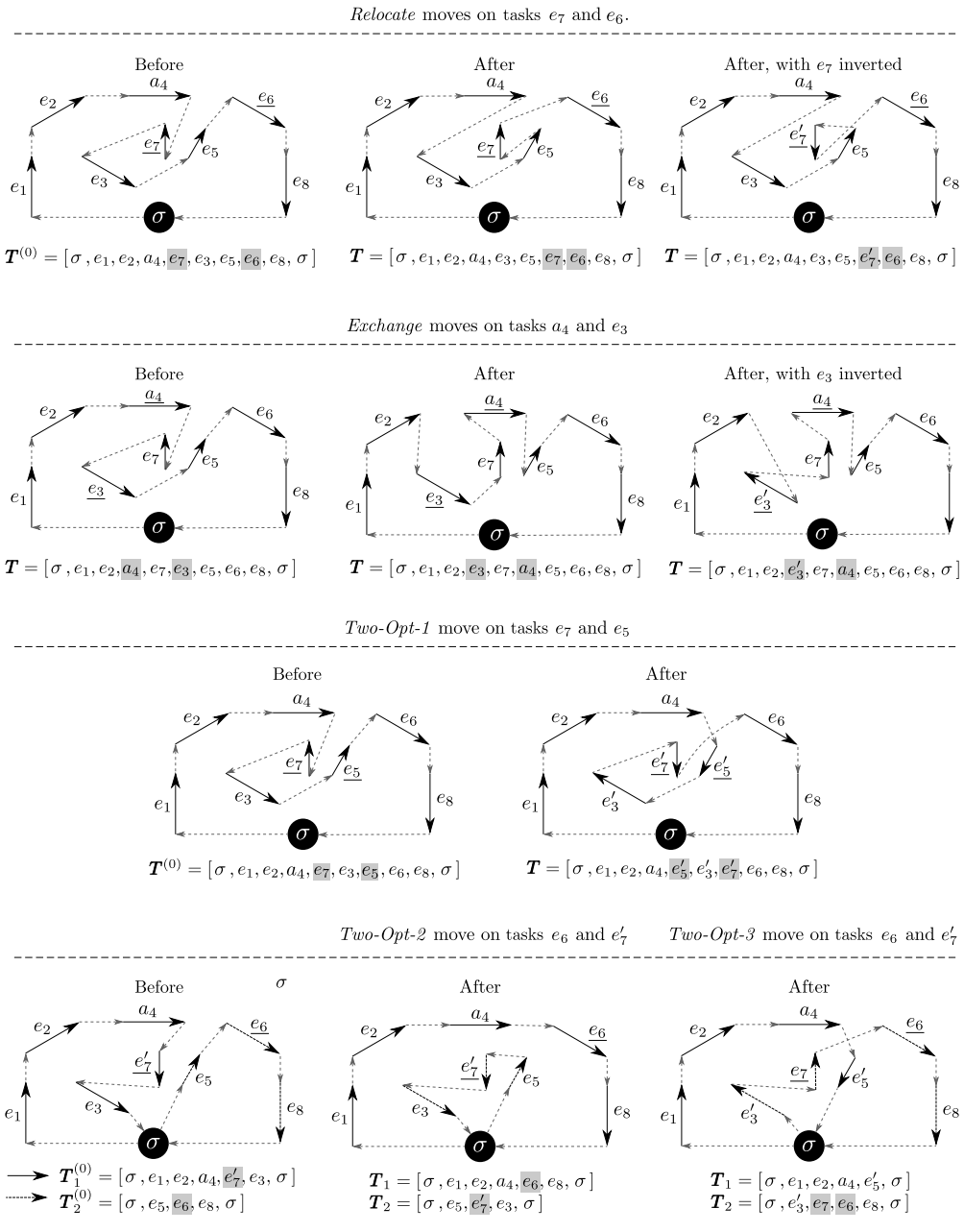


Figure 1: Examples of *relocate*, *exchange* and *two-opt* move operators for the MCARP.

MCARP. For an MCARP route segment to be symmetric, all its tasks must consist of edges, and the shortest path between two tasks must have the same cost in both directions for all consecutive tasks. Unless these conditions hold, the move operators cannot be implemented with the same efficiency since the cost of reversing segments have to be calculated. To overcome this, Lacomme et al. [15] and Belenguer et al. [1] discard *two-opt* moves if the segments contain any arc-task. The example *two-opt-1* move in Figure 1 will thus be allowed, but the *two-opt-3* move will be discarded. For our MCARPTIF LS versions, we only implemented *two-opt-1* and *cross* and did not consider *two-opt-3*. This was done to limit the implementation burden for what is the first study on LS for the MCARPTIF, and we therefore leave the implementation of *two-opt-2* and other move-operators for future work.

The *relocate* move is applied between all tasks u and v in the current solution. For *exchange* and *two-opt* moves, the move between u and v is to the same as the one between v and u, thus only one of the two has to be evaluated. Lastly, *flip* is individually applied to all tasks u in the solution. When a move involves two distinct routes, the resulting changes in route-loads are calculated. Moves that result in the vehicle capacity limit being exceeded are then ignored. There are also several options for choosing which improving move to implement, should there be more than one. When the neighbourhood is searched by evaluating moves one by one, LS may implement the best move found among all those evaluated, or it may implement the first improving move found. The two move strategies are referred to as *best-move* and *first-move*. Both Lacomme et al. [15] and Belenguer et al. [1] use a *first-move* strategy, which is quicker than *best-move*. Since *first-move* terminates the iteration at the first improving move found it only has to partially scan the neighbourhood, unless a local optimum has already been reached. Belenguer et al. [1] further attempt to improve the efficiency of their LS implementation by forcing *two-opt* to discard moves involving asymmetrical route segments. Despite this initiative and the use of *first-move*, they found that the execution time of their Memetic Algorithm is excessive on the biggest of their MCARP test instances. As a remedy, they recommend using advanced acceleration mechanisms to speed-up the LS component of their Memetic Algorithm. In the rest of review, we discuss three such mechanisms and their possible application to the MCARPTIF.

3.2. Nearest neighbour lists

Nearest-Neighbour-Lists was first implemented for the CARP by Beullens et al. [2] to improve the efficiency of their Guided Local Search metaheuristic, which is still one of the fastest heuristics for the CARP [19]. The mechanism has recently been applied by Chen and Hao [3] and Vidal [22], and it is the mechanism that Belenguer et al. [1] suggest to improve their Memetic Algorithm. The lists enable LS to scan a promising subset of the full neighbourhood. For each task u, a nearest neighbour list, $\boldsymbol{N}_u \in \boldsymbol{N}$, is established that contains a fraction, f, of its closest required tasks $v \in \boldsymbol{N}_u$, where $0 < f \leq 1$. The lists are sorted based on the distance from v to u, and the parameter f is user-specified at the beginning of the procedure. The evaluation of moves can then be limited through *Nearest-Neighbour-Lists*. For example, when evaluating a *relocate* move where u is inserted after v, the condition $v \in \boldsymbol{N}_u$ can be enforced, in which case the insert position is limited to closest neighbours of u. The length of the *Nearest-Neighbour-Lists* is controlled through f. A low value of f produces a small subset of the move neighbourhood that can be quickly scanned, but improving moves outside of the neighbourhood subset will not be considered, resulting in LS terminating prematurely. As $f \rightarrow 1$ the full set of moves will be evaluated but without any sort of acceleration taking place. A balance is thus sought between keeping f low enough to accelerate LS, but high enough so that it terminates at high-quality solutions.

Based on the recommendation of Belenguer et al. [1] to use *Nearest-Neighbour-Lists* to accelerate LS for the MCARP, and its successful application to the CARP, we chose to adapt the mechanism for the MCARPTIF. A downside of the mechanism, which we formally investigate, is that there is a trade-off between its acceleration and solution improvement capabilities.

3.3. Static move descriptors

In most LS implementations, after a move is made the entire move-neighbourhood is rescanned, despite the move only modifying a small portion of the solution. For example, when a task is relocated in the same route, all moves not involving that route will be unaffected, yet in most cases, the moves will be rescanned. The same also holds for *first-move* implementations, where towards the end of the search an increasing portion of the neighbourhood has to be searched for the last remaining improving moves, of which only one is made per iteration. To address this inefficiency, Zachariadis and Kiranoudis [28] develop *Static-Move-Descriptors* for the VRP that describe every possible LS move towards a new solution. Importantly, they allow information on all moves to be recorded and reused in following

iterations. During an LS iteration the best move is identified and implemented, and in the following iterations, only moves that involve nodes that were influenced by previously implemented moves are rescanned, and their descriptors updated. The best move is then identified, and the process repeats until a local optimum is reached. *Static-Move-Descriptors* allow LS to return the same local optima as basic LS with a *best-move* strategy. As such, its application does not result in any trade-off on solution quality.

Zachariadis and Kiranoudis [28] test the acceleration mechanism on the VRP and found that on small test instances with less than 400 required nodes, their accelerated LS has a similar execution time of a basic *best-move* version. On larger problems, the accelerated LS was much faster per iteration, exhibiting linearithmic ($n \log n$) growth with problem size, whereas the basic LS heuristic exhibits quadratic growth. On a 1200 required-nodes VRP instance, the basic LS implementation takes about eight times longer per iteration than the accelerated version.

Beullens et al. [2] implement a similar strategy, which they refer to as edge-marking, for the CARP. They further link edge-marking with *Nearest-Neighbour-Lists* and a *first-move* strategy. As such, their LS is not guaranteed to return true local optima with respect to its move operators. Their tests are also limited to the small benchmark instances, making it difficult to predict what effect the acceleration mechanism will have on large MCARPTIF instances. Consequently, we chose to only adapt the *Static-Move-Descriptors* of Zachariadis and Kiranoudis [28] for the MCARPTIF. The adaptation was made easier due to the solution representation, encoding scheme, and move operators of the VRP being similar to those of the MCARPTIF. We further linked *Static-Move-Descriptors* with *Nearest-Neighbour-Lists* and evaluated the solution cost and execution time trade-off of the linked and unlinked versions on large waste collection instances.

3.4. Compounding independent moves

The last acceleration mechanism that we review is *Compound-Independent-Moves*, as applied to the VRP by Ergun et al. [10] and by Dell'Amico et al. [8] to the Mixed Capacitated General Routing Problem (MCGRP). The MCGRP is a generalisation of the MCARP in which vertices as well as arcs and edges have to be serviced on a mixed road network. *Compound-Independent-Moves* is based on the same principle as *Static-Move-Descriptors*. Moves that are not influenced by a previous move are considered independent from that move. All independent improving moves will remain improving regardless of the sequence in which they are implemented, and can, therefore, be made in the same LS iteration. Ergun et al. [10] use this principle to create new neighbourhoods by compounding (combining) smaller independent moves. A series of compounded independent moves, constituting a single super-move, is then made in each iteration, which allows LS to reach local optima in fewer iterations.

There are different methods to determine which independent moves to compound. Dell'Amico et al. [8] use a greedy approach, which we refer to as *Greedy-Compound-Independent-Moves*, that starts with the best move and then continue to the next best improving move that is independent of all previous moves made. LS will then move to the next iteration when no more independent moves are left, after that the full neighbourhood is again scanned. Since *Compounded-Independent-Moves* are seen as a single super-move, this greedy approach may not produce the best compounded move in terms of total improvement. To illustrate, let π_1 be the best move which is dependent on the second and third best moves, π_2 and π_3, with π_2 and π_3 being independent of each other and having a better combined savings than π_1. In this case, making the compounded move of π_2 and π_3 instead of π_1 is a better move in the compounded neighbourhood.

Ergun et al. [10] show that finding the best moves to compound into a super-move is in itself an $\mathscr{NP}$-hard problem, for which they develop a multi-label shortest path algorithm to search the compounded neighbourhood heuristically. Implementing the heuristic is non-trivial, and we leave its application to the MCARPTIF for future work. We instead implemented and tested the *Greedy-Compounded-Independent-Moves* mechanism. We then linked the mechanisms with *Static-Move-Descriptors* as well as *Nearest-Neighbour-Lists*.

4. Basic Local Search for the MCARPTIF

In this section we build on the MCARPTIF solution representation given in Section 2, with key terms summarised in Table 1, and illustrate how one of the move operators can be adapted to the MCARPTIF. Full details on our adaptation of all the move operators to the MCARPTIF can be found in [23]. To make our illustrations more concise

Table 1: MCARPTIF symbols and solution representation summary.

$\boldsymbol{R}$	Set of required tasks
σ	Depot dummy arc
$D(u, v)$	Shortest dead-heading path time from u to v
$\Phi^*(u, v)$	Best IF to visit between u and v
$inv(u)$	Pointer to opposite arc of u, where $inv(u) = 0$ for arc tasks
$\boldsymbol{T}$	MCARPTIF solution
$\boldsymbol{T}_i$	Route i
$\boldsymbol{T}_{i,j}$	Subtrip j of route $\boldsymbol{T}_i$
$T_{i,j,n}$	n^{th} task in subtrip $\boldsymbol{T}_{i,j}$
$\boldsymbol{R}_T$	Set of required tasks currently in solution $\boldsymbol{T}$

we use Φ' as a placeholder to denote the best IF visit between the last and first tasks of consecutive subtrips $\boldsymbol{T}_{i,j}$ and $\boldsymbol{T}_{i,j+1}$, such that $\Phi' = \Phi^*(T_{i,j,|T_{i,j}|-1}, T_{i,j+1,2})$. Routes are then illustrated in the form

$$\boldsymbol{T}_i = [\ldots[\ldots, T_{i,j,|\boldsymbol{T}_{i,j}|-2}, T_{i,j,|\boldsymbol{T}_{i,j}|-1}, \Phi'], [\Phi', T_{i,j+1,2}, T_{i,j+1,3}, \ldots]\ldots].$$

Another modification that we made to the solution representation of Willemse and Joubert [25] is that the last IF and depot visit tasks are included in a separate subtrip at the end of the route, such that

$$\boldsymbol{T}_i = [\ldots, [\ldots, T_{i,j^*-1,k^*-1}, \Phi'], [\Phi', \sigma]],$$

where $j^* = |\boldsymbol{T}_i|$ and $k^* = |\boldsymbol{T}_{i,j-1}|$. This allows all the subtrips, except for the last which is never subjected to moves, to be treated the same by the LS operators. Otherwise, special checks have to be preformed to ensure that tasks are not relocated between the IF and depot.

4.1. Basic LS framework and the relocate move operator

Our basic LS heuristic for the MCARPTIF relied on five move operators that are a subset of those used in [1, 15] and reviewed in Section 3.1 for the MCARP. The operators are *flip*, *relocate*, *exchange* and two versions of *two-opt*, namely *cross*, which evaluates moves between different routes, and *two-opt-1* which evaluates moves in a single route. To analyse its full savings potential, *two-opt-1* considers all route-segment inversions, including asymmetrical segments.

The main extension required for the MCARPTIF was to adapt the MCARP move operators to deal with IFs and the route time-duration constraint. IFs influence cost calculations and the additional constraint has to be checked for move feasibility. To illustrate the adaptations, consider the *relocate* move shown in Figure 2. The MCARPTIF move relocates a task to a different position in the same subtrip, a different subtrip of the same route, or to another route. In our implementation, we chose to make the relocate position the new position of task u. With a relocate position of $T_{l,m,n} = v$, task u is inserted before v such that $T_{l,m,n} = u$, $T_{l,m,n+1} = v$ and $T_{l,m,n+i} = T_{l,m,n+i-1}$ for all $i \in \{1, \ldots, |\boldsymbol{T}_{m,n}| - 1 - n\}$. It should be noted that this version produces an identical neighbourhood to the more commonly used version of relocating task u after v. The move is evaluated between the removal of all tasks in $\boldsymbol{T}$ and their insertions into all the possible new positions.

The cost of the *relocate* move, ΔZ, consists of the cost of removing the task from a subtrip, and the cost of inserting it into a new position. Compared to the MCARP, the removal of tasks next to IF visits can impact the choice of the best IF to visit, and therefore require different cost calculations than with the other tasks. Special cost calculations, relying on $\mu^*(u, v)$ are also required when the insert positions are next to IFs. With reference to the example move in Figure 2, the cost of the move, ΔZ, is calculated as

$$\Delta Z = D(u_1, u_3) - D(u_1, u_2) - D(u_2, u_3) + \mu^*(v_3, u_2) + D(u_2, v_4) - \mu^*(v_3, v_4) \tag{6}$$

There are special calculations when a task is removed from a subtrip servicing only that task, for example when removing task v_6 in Figure 2, in which case the subtrip is removed.

In addition to the new cost calculations for the MCARPTIF, subtrip load-capacity and route-duration feasibility checks are performed, depending on where the task is relocated. If the task is relocated to the same subtrip, demand

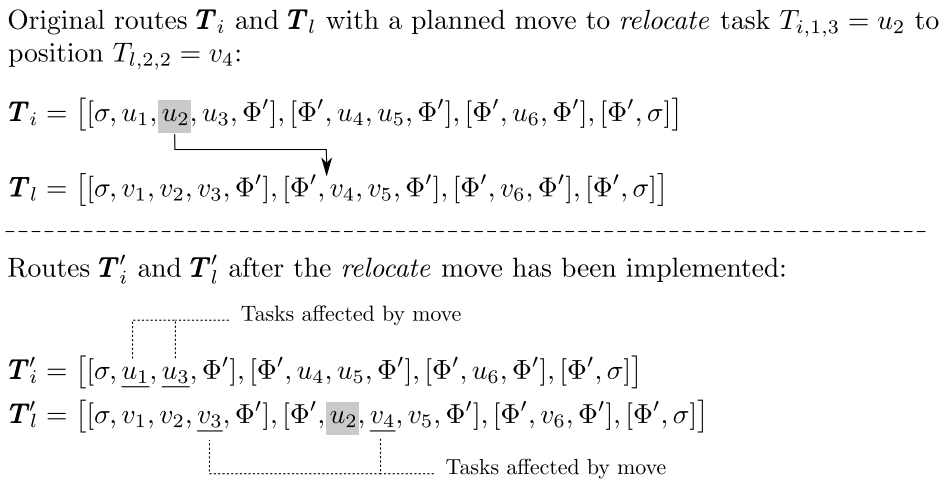


Figure 2: Example of the MCARPTIF *relocate* move where a task is removed from a subtrip, and inserted into another position. The taks can be inserted in the same subtrip, a different subtrip in the same route, or in a subtrip of a different route.

will remain unchanged, and the cost of the route will be reduced, thus no checks are required. If it is relocated to a different subtrip in the same route, then a load-capacity check is performed on the subtrip to which the task is relocated. Lastly, if it is relocated to a different route, then a subtrip load-capacity and route-duration check is performed on the subtrip and route to which the task is relocated.

The *flip*, *exchange*, *cross* and *two-opt-1* move operators were similarly adapted from the MCARP to MCARPTIF. To limit the length of the article, we refer the reader to [23] for the full implementation details of all the move operators including their cost calculations, feasibility checks and technical implementations.

4.2. Extending the move operators

In addition to the five classical move operators, the following extensions to *cross*, *relocate* and *exchange* were considered. The first extension allows *cross* to swap the end portions of subtrips, instead of the end portions of entire routes. When this alternative *cross* move is applied between $T_{i,j,k} = u$ and $T_{l,m,n} = v$, where

$$\boldsymbol{T}_i = [\ldots, \boldsymbol{T}_{i,j-1}, [\ldots, T_{i,j,k-2}, T_{i,j,k-1}, \underline{T_{i,j,k}}, T_{i,j,k+1}, \ldots], \boldsymbol{T}_{i,j+1}, \ldots], \tag{7}$$

$$\boldsymbol{T}_l = [\ldots, \boldsymbol{T}_{l,m-1}, [\ldots, T_{l,m,n-2}, T_{l,m,n-1}, \underline{T_{l,m,n}}, T_{l,m,n+1}, \ldots], \boldsymbol{T}_{l,m+1}, \ldots], \tag{8}$$

the end result would then be

$$\boldsymbol{T}'_i = [\ldots, \boldsymbol{T}_{i,j-1}, [\ldots, T_{i,j,k-2}, T_{i,j,k-1}, \underline{T_{l,m,n}}, T_{l,m,n+1}, \ldots], \boldsymbol{T}_{i,j+1}, \ldots], \tag{9}$$

$$\boldsymbol{T}'_l = [\ldots, \boldsymbol{T}_{l,m-1}, [\ldots, T_{l,m,n-2}, T_{l,m,n-1}, \underline{T_{i,j,k}}, T_{i,j,k+1}, \ldots], \boldsymbol{T}_{l,m+1}, \ldots]. \tag{10}$$

Note that $\boldsymbol{T}_{i,j+1}$ remains in $\boldsymbol{T}'_i$ and $\boldsymbol{T}_{l,m+1}$ in $\boldsymbol{T}'_l$. In our implementations we only evaluated the subtrip *cross* move between $T_{i,j,k} = u$ and $T_{l,m,n} = v$ if $T_{i,j+1}$ and $T_{i,m+1}$ start at the same IF, in which case the cost of the move between $T_{i,j+1,2}$ and $T_{l,m+1,2}$ will be zero. Otherwise, a normal *cross* move between the routes was evaluated. The evaluation was always done in the order of first checking if a feasible subtrip move can be made, otherwise checking if a feasible route move can be made.

The second extension that we implemented was to compound two *cross* moves on the same subtrip. There are cases where two *cross* moves on the same subtrip can be compounded into a single *double-cross* move, without having to recalculate move costs. Let ΔZ_1 be the cost for the *cross* move between tasks $T_{i,j,k_1} = u_1$ and $T_{i,j,n_1} = v_1$, and let ΔZ_2 be the cost of the second move on the same subtrip between $T_{i,j,k_2} = u_2$ and $T_{i,j,n_2} = v_2$. If $k_1 < k_2 - 1$, $k_2 < n_1 - 1$ and

$n_1 < n_2 - 1$ the two *cross* moves can be compounded to produce an exchange between sections $[T_{i,j,k_1}, \ldots, T_{i,j,k_2-1}]$ and $[T_{i,j,n_1}, \ldots, T_{i,j,n_2-1}]$. In this case the original subtrip

$$\boldsymbol{T}_{i,j} = [\ldots, T_{i,j,k_1-1}, \underline{T_{i,j,k_1} \ldots, T_{i,j,k_2-1}}, T_{i,j,k_2}, \ldots, T_{i,j,n_1-1}, \underline{T_{i,j,n_1}, \ldots, T_{i,j,n_2-1}}, T_{i,j,n_2}, \ldots], \tag{11}$$

will become

$$\boldsymbol{T}'_{i,j} = [\ldots, T_{i,j,k_1-1}, \underline{T_{i,j,n_1}, \ldots, T_{i,j,n_2-1}}, T_{i,j,k_2}, \ldots, T_{i,j,n_1-1}, \underline{T_{i,j,k_1}, \ldots, T_{i,j,k_2-1}}, T_{i,j,n_2}, \ldots]. \tag{12}$$

The cost of the *double-cross* move is $\Delta Z_1 + \Delta Z_2$, and it can also be implemented if $k_2 < k_1 - 1$, $k_1 < n_2 - 1$ and $n_2 < n_1$.

The last extension that we implemented was to compound an infeasible improving move with a non-improving move. We applied this for *exchange* and *relocate* between different subtrips on the same route; thereby only the capacity constraint is of concern. A move that violates the capacity constraint of subtrip $\boldsymbol{T}_{l,m}$ can be linked with an independent *relocate* move that removes task v' from $\boldsymbol{T}_{l,m}$. The two moves are compounded if v' can be feasibly inserted into another subtrip, and if its removal frees-up enough capacity in $\boldsymbol{T}_{l,m}$ for the infeasible move to become feasible. This type of move is referred to as an *infeasible-compound* move.

For the *double-cross* moves we used a greedy approach to decide which two moves to compound. *Cross* moves were grouped according to the subtrips on which they are applied, and the moves in each group were sorted from best to worst savings. Starting with the best move, the heuristic would scan the rest of the moves in the subtrip group until one is found that meets all the *double-cross* conditions. If none is found, the process repeats from the second best move in a group. If a move was found, both are implemented. Each subtrip group is scanned through this process. Importantly, this approach simply reuses the existing acceleration mechanisms.

The same approach is followed for *infeasible-compound* moves. First, infeasible subtrip moves are grouped according to subtrips together with complimentary non-improving *relocate* moves that remove tasks from the subtrip. Starting with the best infeasible move, the non-improving complimentary *relocate* moves for that subtrip are scanned from best to worst. When a non-improving move is found that releases enough capacity from the subtrip both are implemented. The process is applied to all subtrip groups.

For our LS implementations, we searched for and applied *double-cross* and *infeasible-compound* only once LS reached a local optimum. If improving *double-cross* or *infeasible-compound* moves were found, LS reverted to the normal search, otherwise it terminated.

Both *double-cross* and *infeasible-compound* further evaluates pairing improving and non-improving moves, as long as the combined moves result in an improvement. The move neighbourhood can thus be extended by evaluating non-improving moves, which can be limited by specifying a threshold saving $\Delta\overline{Z}$. Only moves with $\Delta Z < \Delta\overline{Z}$ are then evaluated. Feasible moves with $\Delta Z < 0$ can be directly implemented, and *double-cross* and *infeasible-compound* moves can be implemented if $\Delta Z_1 + \Delta Z_2 < 0$, where ΔZ_1 and ΔZ_2 are the respective savings of the two compounded moves. The challenge is then to find a threshold saving that improves the local optima at which LS terminates while still keeping it efficient, keeping in mind that a good threshold may be instance specific. In this paper, we formally analysed the move-cost landscape of our LS move operators to find appropriate threshold values.

Additional move operators can also be adapted for the MCARPTIF, such as *double-relocate* and more advanced *two-opt-1* and *cross* moves. It is not uncommon for researchers to use as much as twelve operators [20]. However, care must be taken when extending the operators as it increases the computational time of LS, and as mentioned, basic LS implementations have already been found to be slow on large problem instances. For this reason, *double-cross* and *infeasible-compound* moves were only used in our accelerated implementations that could afford the additional computational time.

5. Acceleration mechanisms for the MCARPTIF

To accelerate our LS implementations, we adapted and ultimately combined three existing acceleration mechanisms, namely *Nearest-Neighbour-Lists*, *Greedy-Compounded-Independent-Moves* and *Static-Move-Descriptors*. The mechanisms were applied to *relocate*, *exchange* and *cross*, as they are easier to incorporate within the mechanisms and, as we will show in the next section, they contribute the most to the total savings obtained through LS on large

MCARPTIF instances. Before presenting the acceleration mechanisms in detail, we first give general LS functions required for their implementation.

All the three acceleration mechanisms require a task focused search, whereby a move is defined between two tasks, u and v. To enable this, we define a mapping function, $T^{-1}(u) = (i, j, k)$, that maps each task $u \in \boldsymbol{R}$ to its current location in the solution such that $T_{i,j,k} = u$. When $inv(u) \neq 0$, the function lets $inv(u)$ point to the same position as u, such that

$$T^{-1}(inv(u)) = T^{-1}(u) = (i, j, k) \tag{13}$$

The mapping function is automatically updated whenever the solution changes. Using *relocate* as an example, moves will be evaluated between all the required tasks, $u \in \boldsymbol{R}$, and all the required tasks currently in the solution, $v \in \boldsymbol{R}_T$. The function lets $inv(u)$ point to the same position as u when $inv(u) \neq 0$ thereby allowing *relocate* to automatically consider task inversion moves. For this reason, *relocate* moves are evaluated for $u \in \boldsymbol{R}$. The reason for only considering insert position for $v \in \boldsymbol{R}_T$ is that if $v \in \boldsymbol{R}$ was considered, it would result in needlessly testing the same insertion positions twice when $inv(v) \neq 0$. A specific *relocate* move in which u is relocated in-front of v is referred to as $relocate(u, v)$; an *exchange* move in which task u is exchanged with task v is referred to as $exchange(u, v)$; and a *cross* move where two routes are crossed at tasks u and v is referred to as $cross(u, v)$. Exchange moves are evaluated between all tasks $u, v \in \boldsymbol{R}$ to allow for the possible inversions of u and v. *Cross* moves are only considered for $u \in \boldsymbol{R}_T$ and $v \in \boldsymbol{R}_T$ since it does not consider task inversions.

When searching move neighbourhoods within each LS iteration, information on all improving moves are stored in an improving move list $\boldsymbol{M}$. Information per improving move $\pi \in \boldsymbol{M}$ include $\pi = (\Delta Z, move_i, u, v)$, where ΔZ is the cost of the move, $move_i$ is the unique identifier of the move, and u and v are the tasks between which the move is applied. Feasibility checks are initially ignored when searching for improving moves. Instead, all the improving moves are returned, and the feasibility checks are performed on a subset of improving candidate moves before their implementation. Using $\boldsymbol{M}$, the best feasible, or multiple feasible moves can be implemented. The conditions and move cost calculations for each move, depending on the location of the tasks in the current solution, can be found in [23].

Nearest-Neighbour-Lists attempt to accelerate LS by reducing the task sets $\boldsymbol{R}$ and $\boldsymbol{R}_T$ between which moves are evaluated, whereas *Greedy-Compounded-Independent-Moves* and *Static-Move-Descriptors* attempt to more efficiently use and update the improving move list $\boldsymbol{M}$. All three mechanisms are described in detail in the rest of the section.

5.1. Nearest neighbour lists

Nearest-Neighbour-Lists are used in each LS iteration to scan a promising subset of the full solution neighbourhood. To illustrate why this may be beneficial, consider the example network shown in Figure 3. Assume that u is serviced together with all its nearest neighbours in a route, and that v is serviced with all its neighbours in a different route. A *relocate* move between tasks u and v would evaluate the move of relocating task u from its current service position to be serviced directly before v. Traveling from v to u and then back to the neighbours of v will significantly increase the amount of deadheading in the route of v. From a route design perspective, it would be ideal for task u to be serviced directly after one of its nearest neighbour tasks. Similarly, if task is to be relocated before v, it would be ideal if the task is one of the nearest neighbours of v. *Nearest-Neighbour-Lists* formally encapsulate this concept and enables LS to ignore unpromising moves. There may, however, be cases where such unpromising moves still improve the solution, in which case LS may terminate before reaching local optima.

The *Nearest-Neighbour-Lists* of $u \in \boldsymbol{R}$ is formally defined as $\boldsymbol{N}_u \subset \boldsymbol{R}/\{u, inv(u)\}$ and it contains its $s = \lceil f \times |\boldsymbol{R}| \rceil$ closest required tasks, where $0 < f \leq 1$ and is user-specified. With our implementation, the lists are sorted in non-decreasing order based on the travel time from u to v, given by $D(u, v)$. The full neighbourhood list is always available and f, which we defined as a global variable, is directly used to limit the move set. Individual tasks in $\boldsymbol{N}_u$ are given as $N_{u,i}$ where $N_{u,1}$ is the nearest neighbour of u.

Finding improving *relocate* moves can be accelerated through the *Nearest-Neighbour-Lists* as shown in Algorithm 1. The algorithm takes as input a set of tasks, $\boldsymbol{R}$, to be relocated to new positions and a set of tasks, $\boldsymbol{R}_T$, to which the tasks will be relocated. The acceleration takes place in lines 3 to 5 and depends on f. If f is small, relatively few relocate positions will be considered, but the risk of the algorithm missing improving moves increases. If $f = 1$, no acceleration will take place and all improving moves will be considered.

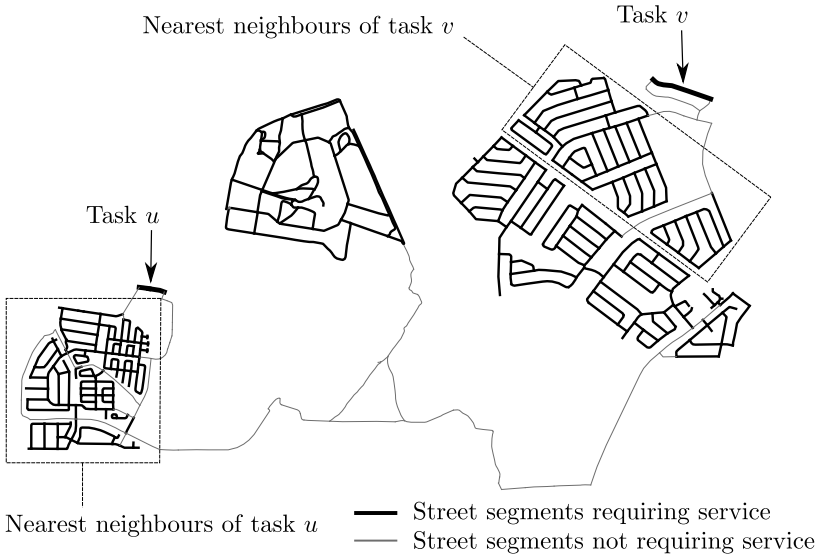


Figure 3: Example of a waste collection area to be serviced. Moves involving u or v should ideally be limited to their respective nearest neighbour tasks.

Algorithm 1: *Find-Relocate-Moves*

Input : Current solution $\boldsymbol{T}$; savings threshold, $\Delta\overline{Z} = 0$; tasks $\boldsymbol{R}$ to be considered for relocation; tasks $\boldsymbol{R}_T$ before which the relocate tasks can be placed; savings list $\boldsymbol{M}$ consisting of information needed to implement moves.
Output: Updated savings list $\boldsymbol{M}$, with information of moves with savings less (better) than $\Delta\overline{Z}$ added to $\boldsymbol{M}$.

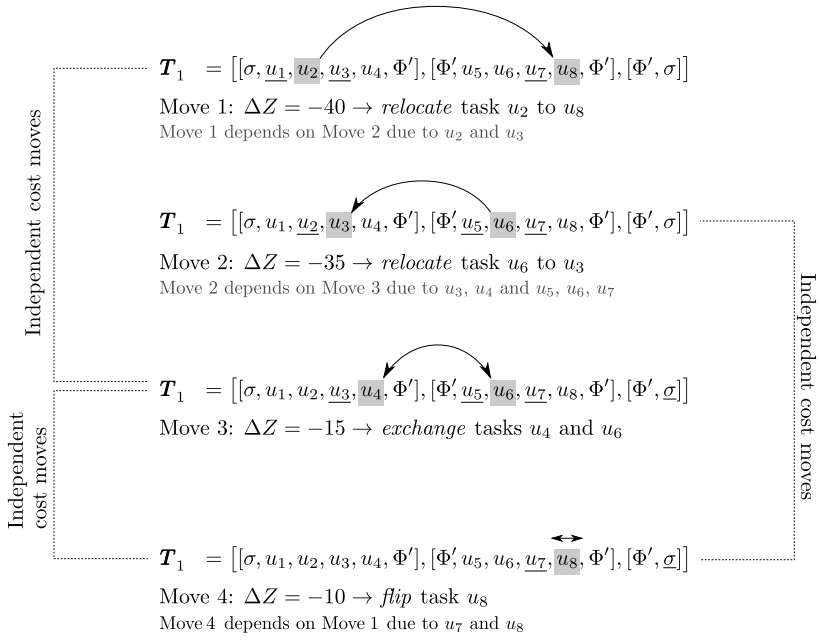
for $u \in \boldsymbol{R}$ **do**
  $(i, j, k) = T^{-1}(u)$;
  $s = \left\lceil f \times |\boldsymbol{R}| \right\rceil$;
  $\boldsymbol{R}'_T = \boldsymbol{R}_T \cap \{N_{u,1}, N_{u,2}, \ldots, N_{u,s}\}$ // the possible relocation positions are limited to tasks that are in the $\boldsymbol{R}_T$ and that are nearest neighbours of u //;
  for $v \in \boldsymbol{R}'_T$ **do**
    $(l, m, n) = T^{-1}(v)$;
    Using $\boldsymbol{T}$ and i, j, k of u, and l, m, n of v, calculate ΔZ for the *relocate* move;
    if $\Delta Z < \Delta\overline{Z}$ **then**
      $\boldsymbol{M} = \boldsymbol{M} \cup \{(\Delta Z, move_i = 1, u, v)\}$;
return ($\boldsymbol{M}$)

Exchange and *cross* require slightly different implementations. Using Figure 3 as an example, assume that v is serviced in a route together with the neighbours of u and that u is serviced together with the neighbours of v. In this case, an *exchange* move between u and v would make sense. However, checking if $u \in \{N_{v,1}, \ldots, N_{v,s}\}$ or $v \in \{N_{u,1}, \ldots, N_{u,s}\}$ would eliminate the move, unless $f \to 1$. For this reason, we instead enforced the membership condition that $v \in \{N_{u_{\text{pre}},1}, \ldots, N_{u_{\text{pre}},s}\}$ where $T^{-1}(u) = (i, j, k)$ and $u_{\text{pre}} = T_{i,j,k-1}$ on *exchange* moves. A *cross* move between $T_{i,j,k} = u$ and $T_{l,m,n} = v$ will relocate task v directly after $T_{i,j,k-1}$. The *Nearest-Neighbour-List* membership condition for *exchange* can thus also be used for *cross*. The full NNL implementations for both moves can be found in Algorithms 2 and 3 in Appendix A.

5.2. Greedily compounding independent moves

In our basic *best-move* LS implementations, all improving moves are compared, and the best move implemented. The idea behind *Independent-Compound-Moves* is to identify independent improving moves and to apply them all simultaneously in a single LS iteration to form a single super-improving move. A move between tasks u_1 and v_1 is considered independent from a move between u_2 and v_2 if implementing either of the moves does not change the move-cost of the other move. The independent moves can then be made together in one LS iteration without having to recalculate their move costs.

Figure 4 shows four possible improving moves on the example route $\boldsymbol{T}_1$ as well as the outcome of compounding two of the independent moves into a single super-move. To determine if moves are independent, consider Move 1



Outcome of compounding Moves 1 and 3 into a single super-move:

$$\begin{aligned}\boldsymbol{T}_1' &= [[\sigma, \underline{u_1}, \underline{u_3}, \underline{u_6}, \Phi'], [, \Phi', \underline{u_5}, \underline{u_4}, \underline{u_7}, \underline{u_2}, \underline{u_8}, \Phi'], [\Phi', \underline{\sigma}]] \\ \Delta Z &= -40 - 15 \\ &= -55\end{aligned}$$

Figure 4: Four possible improving moves on the example route $\boldsymbol{T}_1$. The tasks between which the moves are applied are highlighted in grey, and tasks used to calculate the costs of the moves are underlined.

which involves relocating task u_2 before task u_8. The cost of the move is calculated as

$$\Delta Z = D(u_1, u_3) - D(u_1, u_2) - D(u_2, u_3) + D(u_7, u_2) + D(u_2, u_8) - D(u_7, u_8). \tag{14}$$

Task used in the cost calculation, in addition to u_2 and u_8, are underlined in Figure 4. Move 2 will change the location of u_2 and u_3 relative to each other and is thus dependent on Move 1, the reason being that the cost calculation for Move 2 would no longer be valid after Move 1 is implemented. Move 4 will change the location of u_8 by removing it from the solution and replacing it with $inv(u_8)$. As such, it is also dependent on Move 1. Move 3 does not change the location of any of the tasks used for the cost calculation of Move 1. Should Move 1 be implemented, the cost of Move 3 will remain the same, which is why the moves are considered independent and can be implemented, thereby compounded in the same LS iteration, resulting in a compounded savings of $-40 - 15 = -55$. If Move 2 is implemented instead of Move 1, the cost of Move 4 will be unchanged, so these two moves can be compounded, instead of Moves 1 and 3, for a compounded saving of $-35 - 10 = -45$. The last option is to compound Moves 3 and 4 for a cost of $-15 - 10 = -25$. From all the options, move 1 and 3 should be compounded into a super-move as it gives the best compounded savings.

To identify a sequence of independent moves, we refer to two consecutive tasks, θ_k and θ_{k+1}, in a route as being linked, with the link given as (θ_k, θ_{k+1}). When determining the cost of a move, the links that will change (be broken) and the new links that will be formed through the move are used for the cost calculation. For the *relocate* move in Figure 4 between tasks u_2 and u_8, the links used to calculate ΔZ are those between the underlined and highlighted tasks, specifically (u_1, u_2), (u_2, u_3) and (u_7, u_8). These links are referred to as the move's cost-links. If another move were to break any of the cost-links of a move, the two moves are dependent and cannot be compounded into a super-move. Costs-links can be broken if one of the tasks in the links is removed or replaced, or if a task is inserted in between the linked tasks.

To simplify our notations, the functions $pre(u)$ and $post(u)$ are defined to return the tasks before and after u, respectively, such that:

$$(i, j, k) = T^{-1}(u), \tag{15}$$

$$pre(u) = \begin{cases} T_{i,j,k-1} \text{ if } k > 2 \text{ or } j = 1, \\ T_{i,j-1,|\boldsymbol{T}_{i,j-1}|-1} \text{ otherwise,} \end{cases} \tag{16}$$

$$post(u) = \begin{cases} T_{i,j,k+1} \text{ if } k < |\boldsymbol{T}_{i,j} - 1|, \\ T_{i,j+1,2} \text{ otherwise.} \end{cases} \tag{17}$$

Equations (15) to (17) allow for cost-links to be established between two tasks that are separated by an IF visit. This is necessary when a move involves tasks that are adjacent to IF visits. To check if two moves are independent, one simply needs to check if any of the move's cost-links are broken by the other move. The cost-links for all the move types are shown in Table 2, and the conditions under which each type of move will break a cost-link can found in Table 3. Once $\boldsymbol{M}$ has been populated with information of improving moves, Tables 2 and 3 can be used to determine

Table 2: Cost-links of moves between tasks u and v with $pre(u)$ and $post(u)$ defined in Equations (15) to (17).

Move operator	Cost-links involving u	Cost-links involving v
$Relocate(u, v)$	$(pre(u), u)$ and $(u, post(u))$	$(pre(v), v)$
$Exchange(u, v)$	$(pre(u), u)$ and $(u, post(u))$	$(pre(v), v)$ and $(v, post(v))$
$Cross(u, v)$	$(pre(u), u)$	$(pre(v), v)$

Table 3: Conditions for a move between tasks u and v to break the cost-link (θ_k, θ_{k+1}).

Move operator	Condition for breaking the cost-link (θ_k, θ_{k+1})
$Relocate(u, v)$	If $u = \theta_k$ or $u = \theta_{k+1}$ or $v = \theta_{k+1}$
$Exchange(u, v)$	If $u = \theta_k$ or $u = \theta_{k+1}$ or $v = \theta_k$ or $v = \theta_{k+1}$
$Cross(u, v)$	If $u = \theta_{k+1}$ or $v = \theta_{k+1}$

which of the moves are independent. The next step is then to compound the moves into a single feasible super-move.

When taking capacity and route-duration constraints into consideration, finding the best moves to compound into a feasible super move becomes an $\mathscr{NP}$-Hard problem [10]. To solve the compounding problem, we implemented a greedy heuristic, termed *Greedy-Compound-Independent-Moves*, that identifies and immediately makes feasible independent moves. The heuristic takes as input $\boldsymbol{M}$ which is ordered from the best to the worst improving move. Starting with the first move in the list, the heuristic checks if the move is feasible. If so, the move is implemented. The heuristic then moves to the next improving move in $\boldsymbol{M}$ and checks if it is independent of *all* previous moves that have been implemented in the current LS iteration. This is done by adding all cost-links involved in moves, given in Table 2, to an initially empty set $\boldsymbol{C}$ as the moves are made. When evaluating a move, the cost links that will be broken by the move, as given in Table 2, are checked against the links of previous moves in $\boldsymbol{C}$. If it is independent, the heuristic further checks if the move is feasible. If the move passes both checks, it is implemented. This process repeats until all improving moves in $\boldsymbol{M}$ have been evaluated for implementation.

Sorting $\boldsymbol{M}$ in each iteration adds to the time-complexity of LS. Other non-sort based options can be used to scan $\boldsymbol{M}$, such as scanning $\boldsymbol{M}$ in a random sequence or simply scanning it in the sequence in which the moves were added, similar to a *first-move* strategy. The savings list can also be implemented as a priority-queue whereby it is already sorted when it is scanned. We leave these implementations and their evaluation for future work.

5.3. Static move descriptors

The most computationally expensive component of LS is scanning the move neighbourhood for improving moves. *Nearest-Neighbour-Lists* accelerate LS by reducing the size of the move neighbourhood, whereas *Greedy-Compound-Independent-Moves* attempt to better exploit the information gained from scanning the neighbourhood by implementing multiple improving moves at once. The last acceleration mechanism that we adapt for the MCARPTIF builds on the latter by using *Static-Move-Descriptors*, proposed by Zachariadis and Kiranoudis [28] for the VRP. *Static-Move-Descriptors* are solution independent, and in their full application they describe every possible move and its costs towards a new solution. This allows LS to appropriately record and reuse information gained from scanning the neighbourhood. When a move is implemented, only those moves that are affected by it are rescanned and their descriptors updated.

With our *Nearest-Neighbour-Lists* implementation, essential move information contained in $\pi \in \boldsymbol{M}$ are ΔZ, a unique identifier for the move-type, $move_i$, and tasks u and v involved in the move. Information used for the feasibility checks can also be added, such as pre- and post-tasks of u and v, the change in load to the tasks' subtrip, and the change in cost to the tasks' routes. The move information contained in π, therefore, meets all the requirements of a static descriptor. The only modification needed is then to update $\boldsymbol{M}$ after each LS iteration, instead of repopulating it from scratch. In the first LS iteration, $\boldsymbol{M}$ will be populated with the descriptors of all the improving moves, thereafter it only needs to be updated.

To describe how $\boldsymbol{M}$ can be updated at each iteration, consider an LS implementation that only uses the *relocate* operator. In the first LS iteration the savings list, $\boldsymbol{M}$, of all improving moves can be found and returned using Algorithm 1. The list is then ordered and its first, thus best, feasible improving move will be implemented. After the move is implemented, the next step is to determine which of the descriptors have to be updated. Returning to our *Greedy-Compound-Independent-Moves* implementation, recall that each move has cost-links as defined in Table 2, and each move will break other cost-links. To update $\boldsymbol{M}$ after a move is implemented, all other moves whose cost-links have been broken by the move, as defined in Table 3, have to be rescanned and their descriptors updated. Importantly, *only* these moves have to be updated.

A *relocate* move between u and v has three cost links, $(pre(u), u)$, $(u, post(u))$ and $(pre(v), v)$. If the first implemented move in $\boldsymbol{M}$ between u^* and v^* broke any of these links, the move between u and v has to be rescanned to update its move descriptor. Based on Table 3, a cost-link of the move between u and v would have been broken if $u^* \in \{pre(u), u, post(u), pre(v), v\}$ or if $v^* \in \{u, post(u), v\}$. To update the descriptors, all moves which involve relocating task $pre(u^*)$, u^*, $post(u^*)$, $pre(v^*)$ or v^* have to be rescanned, so too all moves in which a task is inserted before u^*, $post(u^*)$ or v^*. To update the move descriptors, $\boldsymbol{M}$ is scanned and any descriptor with u or $inv(u) \in \{pre(u^*), u^*, post(u^*), pre(v^*), v^*\}$ or $v \in \{u^*, post(u^*), v^*\}$ is removed. Thereafter the descriptors are updated

and inserted back into $\boldsymbol{M}$ using the following equations:

$$\boldsymbol{R}_u = \{pre(u^*), u^*, post(u^*), pre(v^*), v^*\}, \tag{18}$$

$$\boldsymbol{R}_u^{(inv)} = \{inv(u') : u' \in \boldsymbol{R}_u \text{ and } inv(u') \neq 0\}, \tag{19}$$

$$\boldsymbol{R}_v = \{u^*, post(u^*), post(v^*)\}, \tag{20}$$

and by calling Algorithm 1 as follows:

$$\boldsymbol{M}' = \textit{Find-Relocate-Moves}(\Delta\overline{Z}, \boldsymbol{R}_u \cup \boldsymbol{R}_u^{(inv)}, \boldsymbol{R}_T, \boldsymbol{M}), \tag{21}$$

$$\boldsymbol{M}'' = \textit{Find-Relocate-Moves}(\Delta\overline{Z}, \boldsymbol{R}/\boldsymbol{R}_u \cup \boldsymbol{R}_u^{(inv)}, \boldsymbol{R}_v, \boldsymbol{M}'), \tag{22}$$

$$\boldsymbol{M} = \boldsymbol{M}''. \tag{23}$$

Algorithm 1 is called twice to find improving *relocate* moves, first between the subset $\boldsymbol{R}_u$ and $\boldsymbol{R}_T$, and then between $\boldsymbol{R}/\boldsymbol{R}_u$ and $\boldsymbol{R}_v$. *Nearest-Neighbour-Lists* can also be activated by setting $f < 1$ to further accelerate the search. The first time LS searches for improving *relocate* moves, the full neighbourhood is scanned and $\boldsymbol{M}$ is returned using

$$\boldsymbol{M} = \textit{Find-Relocate-Moves}(\Delta\overline{Z}, \boldsymbol{R}, \boldsymbol{R}_T, \boldsymbol{M}), \tag{24}$$

which takes $O(|\boldsymbol{R}| \times |\boldsymbol{R}_T|)$. Thereafter, the neighbourhood is scanned using Equations (21) and (22) in $O(|\boldsymbol{R}| + |\boldsymbol{R}_T|)$.

Since both *Static-Move-Descriptors* and *Greedy-Compound-Independent-Moves* rely on move independence, we use the set $\boldsymbol{C}$, to store the tasks of cost-links of implemented moves. When a move is implemented, its cost-link tasks are added to $\boldsymbol{C}$. The set is used to determine if candidate moves are independent of all the moves already implemented in the current LS iteration, and are also used to update the savings list. By continuously updating the set, all independent moves can be implemented using *Greedy-Compound-Independent-Moves*, after which all tasks affected by the implemented ones are updated. This allows for *Static-Move-Descriptors* and *Greedy-Compound-Independent-Moves* to be combined. Lastly, when updating affected moves, the NNL algorithms can be used to further limit the moves that are updated and added to $\boldsymbol{M}$. A technical description of the acceleration mechanisms and their combined usage can be found in Algorithm 1 in Section 6.5 and in Algorithms 2 to 7 in Appendix A.

6. Computational results

The aim of the paper is to develop efficient LS heuristics for the MCARPTIF that can be extended to or used in metaheuristics applications for the problem. In the previous sections we presented basic and more advanced LS mechanisms and different neighbourhood structures that can be combined to form different LS setups. In this section we present results of computational tests on the different setups.

A total of 20 different LS setups, summarised in Table 4, were implemented and tested. The first four setups used the basic LS setup, without any of the acceleration mechanisms, and either the full or reduced move operators, in conjunction with either the *best-move* or *first-move* strategy. The full move operators consisted of *relocate*, *cross*, *exchange*, *flip* and *two-opt-1* moves. The reduced move operators consisted of *relocate*, *cross* and *exchange*. The remaining sixteen accelerated setups employed the advanced acceleration mechanisms with two options for available move operators and two options for move-strategies. The two move operator options were the reduced move operators, consisting of *relocate*, *cross* and *exchange*, and extended move operators in which *double-cross* and *infeasible-compound* were also applied. The two move strategy options were the *best-move* strategy and the *Greedily-Compounding-Independent-Moves* heuristic. The four combinations of the accelerated setups were further linked with four different nearest neighbourhood levels of $f \in \{0.25, 0.5, 0.75, 1\}$, thus resulting in a total of sixteen unique setups.

Tests were predominantly performed on the *Cen-IF*, *Act-IF* and *Lpr-IF* benchmark sets that cover a range of realistic waste collection instances. The three *Cen-IF* instances, with 1012, 2519 and 2755 required tasks and edges, respectively, represent some of the largest CARP type instances currently available. Tests were also performed on the *mval-IF-3L* benchmark set to make a comparison between the performance of the LS setups on waste instances and the smaller randomly generated *mval-IF-3L* instances. All benchmark sets are discussed in detail in [24, 25] and are available in [26]. Additional tests were also performed on the classical *Lpr* and *mval* instances for the MCARP, available from `https://www.uv.es/belengue/mcarp/`. These tests allow our implementations to be compared

Table 4: Different Local Search setups tested for the MCARPTIF, and where their components and evaluation are featured in the paper.

Setup	Fraction, f, scanned	Move operators	Move strategy	Setup name	Acronym	Paper section reference	
						Components	Evaluation
Basic (B)	$f = 1$	Full (F)	Best move (B)	Basic-Full-Best	BFB	4.1	6.2, 6.7
			First move (F)	Best-Full-First	BFF		
		Reduced (R)	Best move (B)	Basic-Reduced-Best	BRB		
			First move (F)	Best-Reduced-First	BRF		
Accelerated with SMD (A)	$f = \{0.25, 0.5, 0.75, 1\}$	Reduced (R)	Best move (B)	Accelerated-Reduced-Best	ARB-f	4.1, 5.1, 5.3	6.3, 6.7
			GCIM (G)	Accelerated-Reduced-Greedy	ARG-f	4.1, 5.1–5.3	6.4, 6.5, 6.7
		Extended (E)	Best move (B)	Accelerated-Extended-Best	AEB-f	4.1, 4.2, 5.1, 5.3	6.7
			GCIM (G)	Accelerated-Extended-Greedy	AEG-f	4.1, 4.2, 5.1–5.3	6.6, 6.7

Note: Accelerated setups are linked with *Static-Move-Descriptors* (SMD), *Nearest-Neighbour-Lists* with f nearest neighbours, and can be linked with *Greedily-Compounding-Independent-Moves* (GCIM); for setups where f is not specified the level $f = 1$ was used. For the reduced move operators, *relocate*, *cross* and *exchange* are applied. For the extended move operators, *double-cross* and *infeasible-compound* are also applied. For the full move operators, in addition to the reduced move operators, *flip* and *two-opt-1* are also applied.

against future LS implementation. A direct comparison on MCARP instances between our accelerated LS setups and the existing standard LS implementations is not currently possible, since researchers limit tests to the final LS based metaheuristics without reporting directly on the performance of their LS implementations. As discussed in Section 1, we consider a full study focussing on accelerated LS setups as well as their performance in metaheuristics to be beyond the scope of this paper. For preliminary results for the full study, including tests on the MCARP, we refer the reader to [23].

To test the LS setups, three different starting solutions were generated per instance using the *Path-Scanning*, *Improved-Merge* and *Efficient-Route-Cluster* deterministic constructive heuristics that we developed in [25]. The efficiencies of the LS setups were evaluated by measuring the CPU time, in seconds, required to reach local optima on the different initial solutions, and the improvement capabilities evaluated by calculating the fractional cost improvement made by LS to the initial solution. This measurement is given by ΔZ^f_{LS}, and calculated as

$$\Delta Z^f_{\text{LS}} = \frac{Z(\boldsymbol{T}^{(0)}) - Z(\boldsymbol{T}^{(t)})}{Z(\boldsymbol{T}^{(0)})}, \tag{25}$$

where $Z(\boldsymbol{T}^{(0)})$ is the cost of the initial solution and $Z(\boldsymbol{T}^{(t)})$ is the cost of the local optimum solution returned by the LS setup.

The tests were conducted in four phases. In the first phase, the move operators were individually analysed and compared to identify redundant operators and find a move-operator search sequence for the *first-move* strategy. In the second phase, the four basic LS setups were compared to analyse their computational efficiency on large problem instances. In the third phase, we tested the effect of the acceleration mechanisms on LS, which showed that in most cases there is a trade-off between the solution quality and computational efficiency of the setups. Therefore, in the fourth and last phase, all the setups were directly compared to identify dominated setups that are both slower and produce worse solutions than other setups, and are therefore dominated by other setups. The results were then used to choose a non-dominated setup that can be extended to or incorporated into metaheuristics solution methods for the MCARPTIF in future work.

All LS algorithms and procedures were programmed in Python version 2.7, with critical procedures optimised using Cython version 0.17.1. Experiments were run on a Dell PowerEdge R910 4U Rack Server with 128GB RAM with four Intel Xeon E7540 processors. Experiments were run without using programmatic multi-threading or multiple processors. A full results table, with the execution time and final cost of each LS setup per problem instance and initial solution, is available as online supplementary material (see Appendix B).

6.1. Move operator analysis

To individually evaluate and compare the move operators we calculated the fractional cost improvement made by Basic-Full-Best to the initial solution. Basic-Full-Best uses a move neighbourhood consisting of *flip*, *relocate*,

exchange, *cross* and *two-opt-1* and the *best-move* strategy. For each move type we measured the savings that resulted from its moves over the course of an LS run. Next, we calculated the contribution of the moves to ΔZ^f. For example, if Basic-Full-Best improved a solution from $Z(\boldsymbol{T}^{(0)}) = 10\,000$ to $Z(\boldsymbol{T}^{(t)}) = 9000$ and only *relocate* and *exchange* moves were made, its fractional cost improvement would be $\Delta Z^f = 0.1$. If during the search, twelve *relocate* moves were made which resulted in a combined savings of 200, the contribution of *relocate* to ΔZ^f is calculated and reported as $\frac{200}{10\,000} = 0.02$. The contribution of *exchange* is then 0.08.

The contributions of the move-operators to LS over the three initial solutions per instance are shown in Figure 5. On all instances *relocate* made the biggest contribution. The contributions of the other operators depended on the

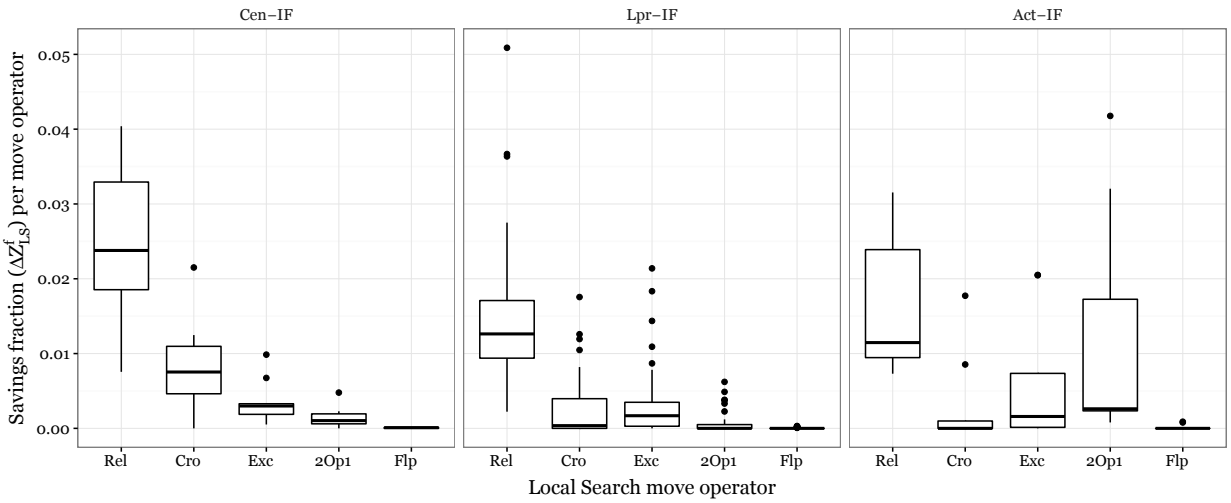


Figure 5: Cost saving contributions on waste collection benchmark sets of *Relocate* (Rel), *Cross* (cro), *Exchange* (Exc), *Two-Opt-1* (2Opt1) and *Flip* (Flp) move operators within the Basic-Full-Best Local Search implementation.

instance set. *Cross* made the second biggest contribution on *Cen-IF* but made little impact on *Act-IF* where *exchange* and *two-opt-1* made much larger contributions. This may be due to the number of required vehicles for the different instances. The *Act-IF* instances require between one and three vehicles, which limits the number of possible *cross* moves to evaluate. The instances are also undirected resulting in the route segment reversals of *two-opt* having smaller cost-changes compared to reversals on mixed road instances. *Flip* contributed little to total LS savings, which may be due to *relocate* and *exchange* automatically inverting tasks if the inversion produces a better move.

To analyse move-operator efficiency, we measured the average time required to scan each move-operator's neighbourhood in a Basic-Full-Best iteration. The average times, per iteration, for the different operators to scan their respective neighbourhoods and return the best move are shown in Figure 6. The three main move operators, *relocate*, *exchange* and *cross* exhibit quadratic growth. This is due to their moves being applied between all tasks in $\boldsymbol{T}^{(t)}$. *Exchange* had the longest execution time per iteration, yet its savings contribution was low in comparison to *relocate*. Although not considered in this paper, it should be interesting to determine what impact its elimination will have on the efficiency of LS. The execution time of *two-opt-1* increased linearly since it only focusses on one route at a time. *Flip* is also very efficient, but as shown in Figure 5, it contributes little to total LS savings.

The aim of the move-operator analysis was to rank the operators for their application within *first-move* and to identify elimination candidates. Since the focus of the paper is on large waste collection instances, we prioritised the results on *Cen-IF* and *Lpr-IF* over those on *Act-IF* and ranked the move-operators in the order shown in Figure 5. The highest ranking operator is thus *relocate*, and the lowest is *flip*. To identify operators for elimination, their ease of implementation was also considered, particularly for the advanced LS acceleration mechanisms. Here we identified *two-opt-1*, due to its route segment reversal, and *flip*, because of its limited impact, as elimination candidates, and adapted *relocate*, *exchange* and *cross* for the accelerated LS setups. Their elimination was also used to improve the efficiency of the basic LS setups by allowing the *best-move* and *first-move* setups to scan a reduced neighbourhood.

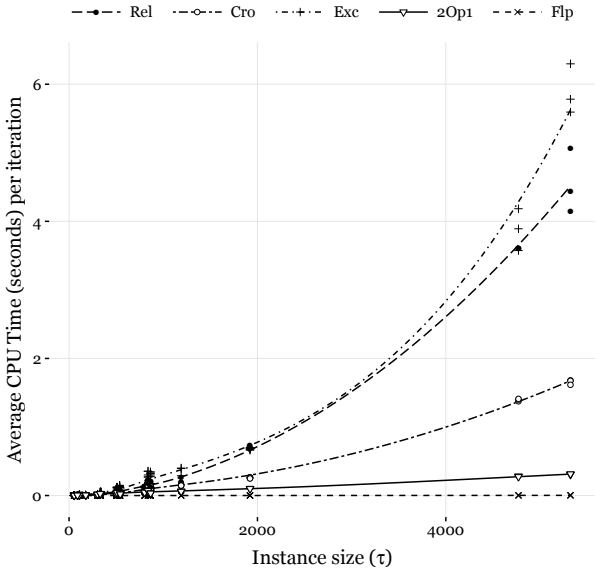


Figure 6: Average time required per iteration by *Relocate* (Rel), *Cross* (cro), *Exchange* (Exc), *Two-Opt-1* (2Opt1) and *Flip* (Flp) to find and return its best improving move within the Basic-Full-Best Local Search implementation on waste collection benchmark sets.

6.2. Best-move and first-move local search analysis

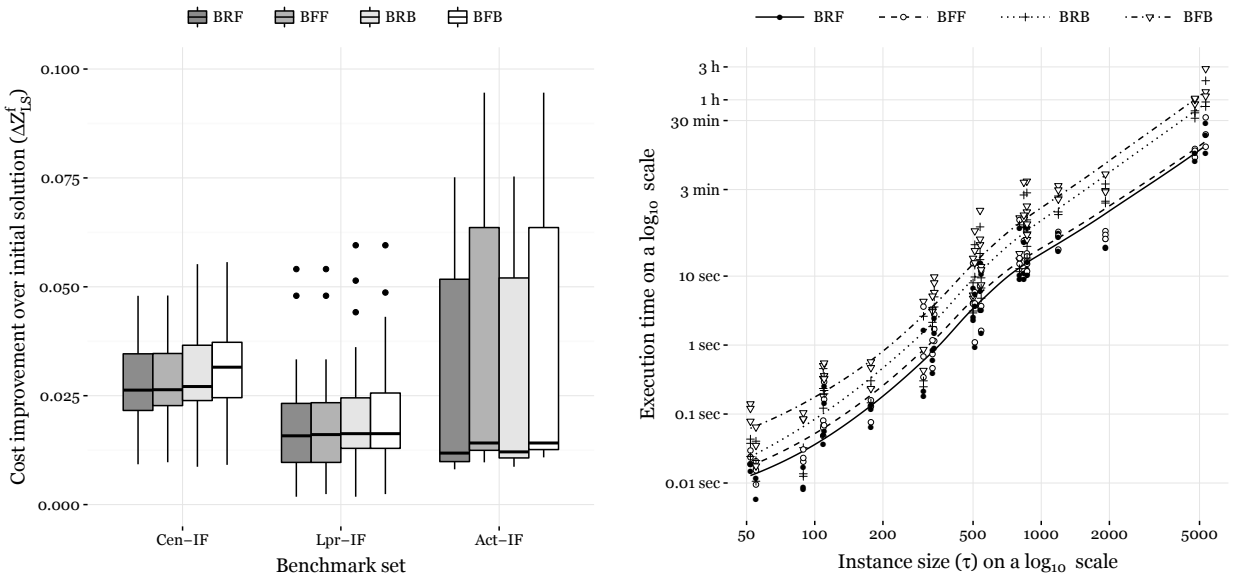
To compare the basic *best-move* and *first-move* implementations, tests were performed on the Basic-Full-First (BFF), Basic-Full-Best (BFB), and Basic-Reduced-First (BRF) and Basic-Reduced-Best (BRB) setups, defined in Table 4. Recall that BRF and BRB use only the *relocate*, *exchange* and *cross* operators, whereas BFF and BFB use the same operators as well as *flip* and *two-opt-1*. The sequence in which the move operators was applied by BRF was *relocate*, *cross* and *exchange*, with the addition of *two-opt-1* and *flip*, in that order, for BFF. The aim of the tests was to evaluate the computational efficiency of the basic setups on large problem instances and to test the impact of using the reduced move neighbourhood.

The cost savings and computational times of all four setups on the three starting solutions per waste collection instance are shown in Figure 7. As shown in Figure 7a, on the *Cen-IF* and *Lpr-IF* instances, the cost savings obtained through the two *first-move* setups were less than the *best-move* setups, with BRF performing the worst. On *Act-IF*, BFF performed better than BRB since it evaluates *two-opt-1* moves, which, as discussed earlier, is a major contributor to savings on these instances. The *best-move* strategy produced better local optima than *first-move*, as did the setups using the full versus reduced neighbourhoods.

The computational times of each setup to reach local optima are shown in Figure 7b. On large problem instances, the *first-move* setups are significantly faster than the *best-move* setups, with BRF being the most efficient. On the largest *Cen-IF* instances, BFF and Basic-Reduced-First took at most thirty-minutes to reach local optima, whereas BFB and BRB took more than 45 minutes and up to three hours. The *first-move* setups are thus more efficient, but their increased speed comes at a trade-off in solution quality. Despite its better efficiency, BRF took between five and thirty-minutes on the largest instances to improve an initial solution to its local optimum. In situations where LS has to be called numerous times, or when short time-limits are imposed, the setup may still be too slow. The test results are therefore consistent with the findings of Belenguer et al. [1] on the MCARP and support their recommendation that more advanced accelerated LS setups be developed and used on large instances.

6.3. Static-Move-Descriptors analysis

The aim of the second round of tests was to evaluate the impact of the more advanced acceleration mechanisms on LS as well as using a slightly extended move neighbourhood. Unlike *first-move* over *best-move*, an advantage of *Static-Move-Descriptors* is that it should not affect solution quality. It may produce slightly different results if there



(a) Fraction by which the LS setups improved the cost of the initial solutions.

(b) CPU times, in seconds, and trend-lines of the LS setups versus problem instance size $\tau = |\boldsymbol{R}|$.

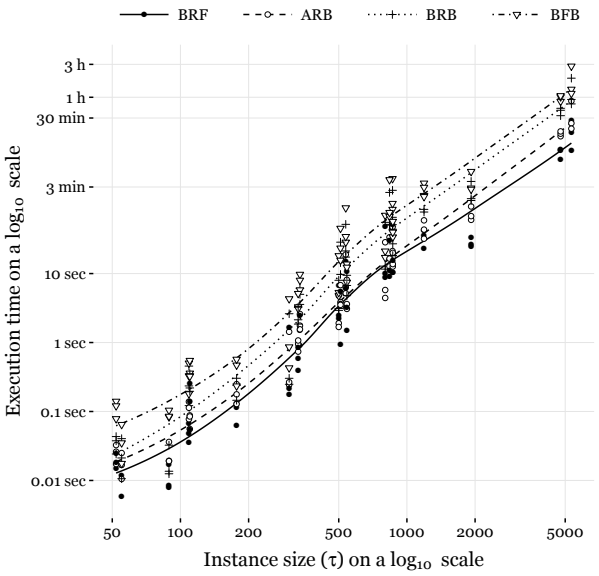
Figure 7: Comparison of Basic-Full-Best (BFB), Basic-Reduced-Best (BRB), Basic-Full-First (BFF) and Basic-Reduced-First (BRF) setups on waste collection benchmark sets.

are tied best-moves, with different tied moves leading to different optima. To evaluate *Static-Move-Descriptors*, the Accelerated-Reduced-Best (ARB) setup was tested against BFB, BRB and BRF, the latter being the most efficient setup from the previous rounds of tests. Results for the setups on the waste collection instances are shown in Figure 8. The execution times of the setups are shown in Figure 8a which confirms that ARB is quicker than BRB and BFB, particularly on the large *Cen-IF* instances. On the large instances, the execution time of ARB was about half-that of BRB. BRB had better cost-savings than ARB. Since both use the same move operators, this is attributed to the setups deciding on different tied-best moves to implement, and we consider the cost-savings differences to be incidental. The acceleration effect of the *Static-Move-Descriptors* was not as significant as those observed by Zachariadis and Kiranoudis [28] on similarly sized VRP instances, indicating that there is room for improvement. One such improvement, which Zachariadis and Kiranoudis [28] found to be critical for their application, is in the use of priority-queues to avoid having to sort the savings list at the start of each LS iteration. All our accelerated LS setups would benefit from this improvement, which we leave for future work.

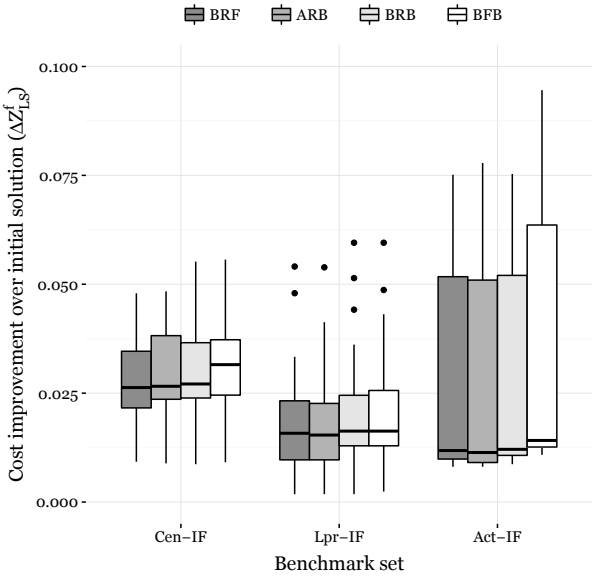
Despite its better efficiency compared to BRB, ARB was still slower than BRF, which indicates that our *Static-Move-Descriptors* implementation, on its own, is insufficient when dealing with large instances. As such, it was linked and tested with *Greedy-Compound-Independent-Moves* and *Nearest-Neighbour-Lists*.

6.4. Greedy-Compound-Independent-Moves analysis

The second acceleration mechanism that we evaluated was *Greedy-Compound-Independent-Moves*. The mechanism was combined with *Static-Move-Descriptors*, with the combined setup referred to as Accelerate-Reduced-Greedy (ARG), and compared against Basic-Reduced-First (BRF), Accelerate-Reduced-Best (ARB) and Basic-Full-Best (BFB) on the waste collection instances. Results of the tests are shown in Figure 9. As shown in Figure 9a, the computational time of ARG is much lower than the times of the other setups, especially on large instances where it took close to three minutes to reach local optima on all but one of the largest *Cen-IF* instances. The difference in computational times between ARG and the other three LS setups increases with problem size, indicating that ARG has better run-time scalability with instance size.

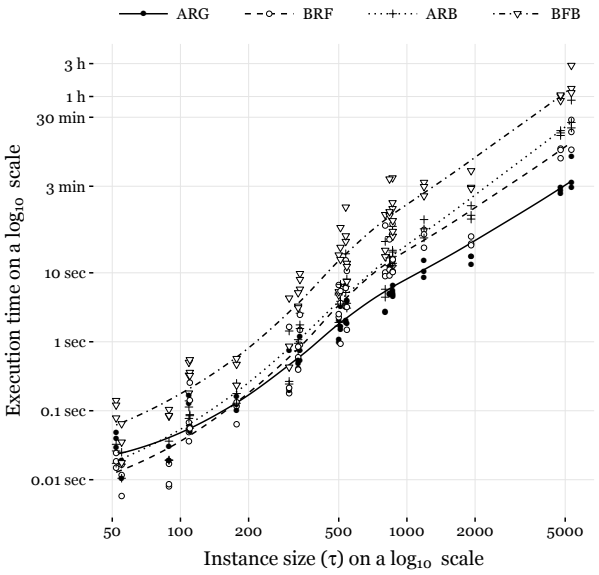


(a) CPU times, in seconds, and trend-lines of the LS setups versus problem instance size $\tau = |\boldsymbol{R}|$.

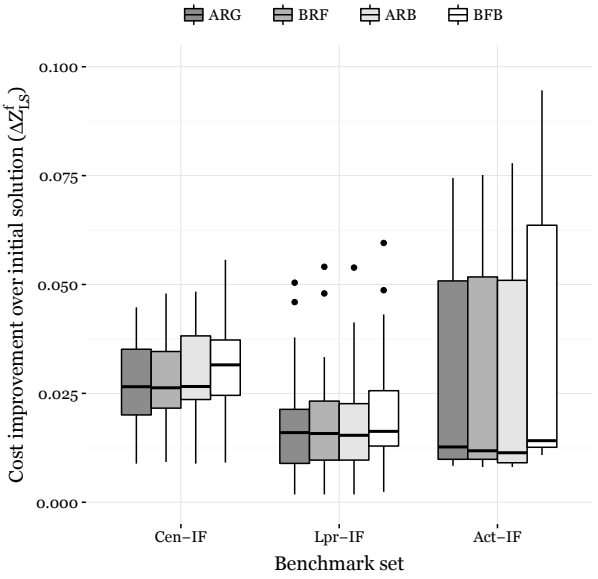


(b) Fraction by which the setups improved initial solutions.

Figure 8: Comparison of Basic-Reduced-First (BRF), Accelerated-Reduced-Best (ARB), Basic-Reduced-Best (BRB) and Basic-Full-Best (BFB) setups on waste collection benchmark sets.



(a) CPU times, in seconds, and trend-lines of the LS setups versus problem instance size $\tau = |\boldsymbol{R}|$.



(b) Fraction by which the LS setups improved initial solutions.

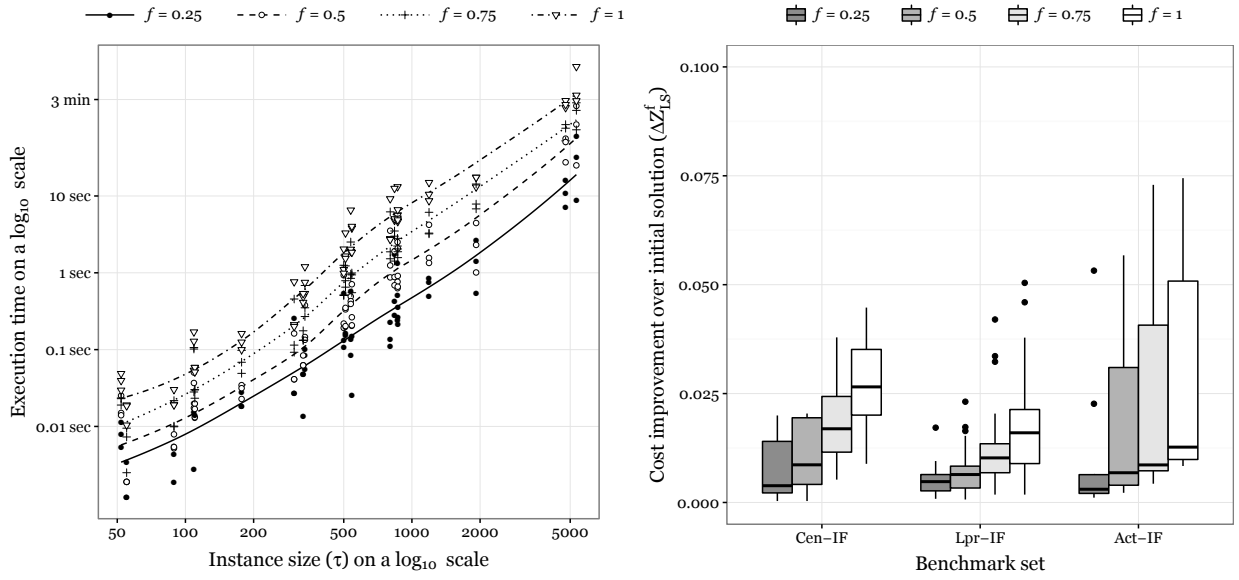
Figure 9: Comparison of Accelerated-Reduced-Greedy (ARG), Basic-Reduced-First (BRF), Accelerated-Reduced-Best (ARB) and Basic-Full-Best (BFB) setups on the waste collection benchmark sets.

As shown in Figure 9b, the improvement of ARG is very similar to that of ARB, outperforming the setup on *Cen-IF* and *Act-IF*. ARB had slightly better improvements on *Cen-IF*, saving it from being completely dominated. ARG was also close to dominating Basic-Reduced-First, being quicker on instances with more than 150 required tasks and edges and producing slightly better solutions.

The solution quality and efficiency of ARG on large instances are promising, but a 3-minute execution time may still be too long for certain applications. It was therefore combined with the last acceleration mechanism from this paper, *Nearest-Neighbour-Lists*.

6.5. Nearest-Neighbour-Lists analysis

Nearest-Neighbour-Lists reduce the move neighbourhood by limiting moves between tasks to a fraction f of the closest neighbours. For the computational tests ARG was tested at three levels, namely $f \in \{0.25, 0.5, 0.75\}$, and compared against the previously tested setups with a full move neighbourhood, i.e. $f = 1$. Results for the setups are shown in Figure 10. As shown in Figure 10a, reducing the move neighbourhood significantly reduced the



(a) CPU times, in seconds, and trend-lines of the LS setups versus problem instance size $\tau = |\boldsymbol{R}|$.

(b) Fraction by which the LS setups improved initial solutions.

Figure 10: Comparison of the Accelerated-Reduced-Greedy setup at four *Nearest-Neighbour-Lists* f-levels on waste collection benchmark sets

computational times of the setup. At $f = 0.25$, the setup took at most 80 seconds on the large *Cen-IF* instances, and in one case less than 10 seconds, to reach local optima. However, as shown in Figure 10b, the reduction in computational times comes at a price, with the coinciding savings of the setups being inversely correlated to f. Linking *Nearest-Neighbour-Lists* with *Static-Move-Descriptors* and *Greedy-Compound-Independent-Moves* allows LS to be used under short execution time-limits, but it also reduces solution quality.

6.6. Extended move operator analysis

The last improvement that we tested was to allow the accelerated setups to use extended move operators searching for improving moves. In addition to the *relocate*, *exchange* and *cross* moves, the extended setups also evaluated *double-cross* and *infeasible-compound* moves. The extended-moves combine an improving move, which cannot be implemented on its own, with an independent complimentary move. The complimentary move does not have to be an improving move, as long as the combined moves result in an improvement on the current solution. Before testing the extended operators, an appropriate cost-threshold had to be determined. Recall that the savings list is sorted and scanned in each iteration to identify complimentary moves to link with infeasible improving moves. To keep these

operations efficient, only moves with savings below a user-set threshold can be included in the savings list. The challenge is then to determine good cost-thresholds for the setups, keeping in mind that they may be benchmark and even instance specific.

To establish a move-cost threshold, we analysed the move-cost landscape of LS with a reduced neighbourhood of *relocate*, *exchange* and *cross* at the first LS iteration. For each initial solution, LS was called but terminated before an improving move was made. Instead, the number of improving and worsening moves were recorded, as well as the number of neutral moves with zero cost. Figure 11 shows the results of the tests on the waste collection sets. The number of available moves increases quadratically with problem size, with worsening moves being the most

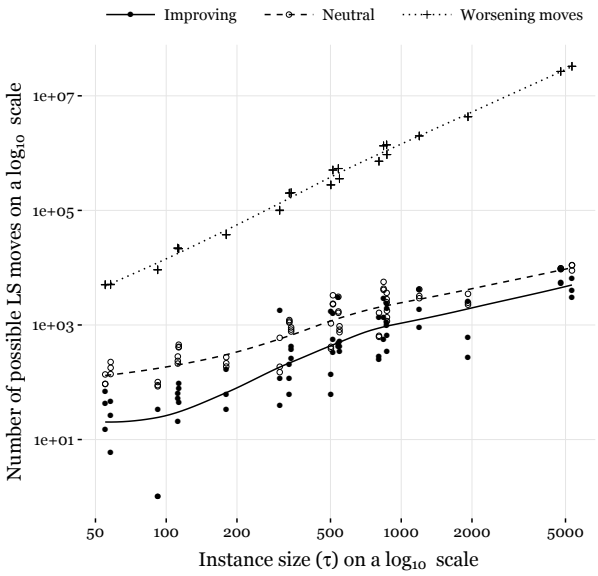
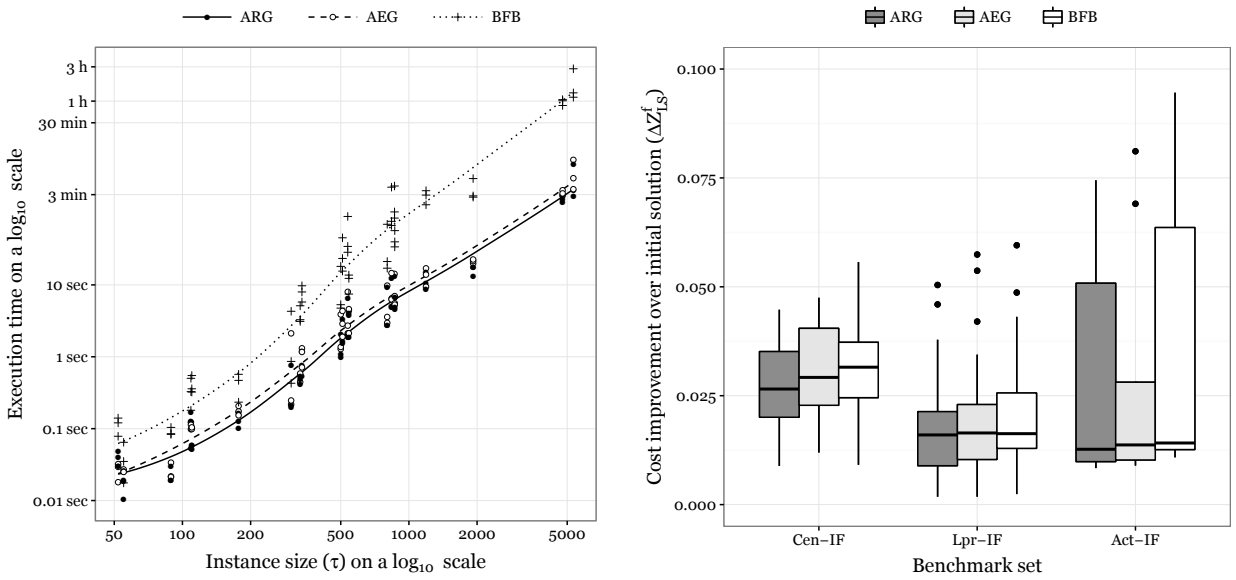


Figure 11: Move landscape analysis at the first iteration of LS with a reduced neighbourhood on waste collection benchmark sets.

prevalent. Of interest is that the number of worsening moves increases at a higher rate than improving and neutral moves as the problem size increases. Through informal tests we found that this holds for the individual move operators as well. Using more move operators on large instances may, therefore, introduce a disproportionate amount of non-improving moves, which will make LS slower without improving solution quality. For future work, we recommend systematically testing this phenomenon by comparing the impact of additional move operators on small versus large problem instances.

Of relevance to setting the cost-thresholds is the high number of available neutral moves, which in most cases outnumber the improving moves. This may be a key feature of waste collection instances and worth exploiting. In an MCARPTIF solution, required tasks are often dead-headed in routes. When a move results in the task being serviced instead of dead-headed in a particular route, the cost of the move is zero. For the extended move neighbourhoods, we chose to exploit this characteristic by setting the move-cost threshold $\Delta Z = 1$, thereby allowing LS only to return improving and neutral moves. An advantage of this approach is that it keeps the savings-list relatively short by eliminating the large portion of non-improving moves. Importantly, a unique threshold does not have to be determined for each problem instance. Regardless of whether the move costs are between $[-10, 10]$ or $[-10000, 10000]$, neutral moves always have zero cost, and as shown in Figure 11 there are usually a significant number of neutral moves available.

To evaluate the impact of the extended move operators, the accelerated LS setup with the extended operators and using the greedy-compound moves, referred to as Accelerated-Extended-Greedy (AEG), was tested on the three starting solutions for the waste collection sets. The results for the setups in comparison to ARG and BFB are shown in Figure 12. The extended move operators increased the computational time of both setups, more so on the mid-sized *Lpr-IF* instances where $75 < |\boldsymbol{R}| < 500$ (Figure 12a). On the large *Cen-IF* instances the computational times required



(a) CPU times, in seconds, and trend-lines of the LS setups versus problem instance size $\tau = |\boldsymbol{R}|$.

(b) Fraction by which the LS setups improved initial solutions.

Figure 12: Comparison of Accelerated-Reduced-Greedy (ARG), Accelerated-Extended-Greedy (AEG) and Accelerated-Extended-Best (AEB) local search setups on waste collection benchmark sets.

to find local optima were relatively close. Better cost savings were obtained through the extended operators, especially on the *Cen-IF* instances (Figure 12b). True to its purpose, the extended operators allowed the LS setups to reach better local optima, and as expected, it increased the computational times of the setups in doing so, although not by much.

6.7. Domination analysis

The second phase of tests on the acceleration mechanisms showed that they have the desired impact of improving the efficiency of LS and that their solutions can be improved through the extended move operators at a slight increase in computational time. The other setups tested resulted in a similar trade-off, by either being quicker than other setups, but producing lower quality solutions, or vice-versa. In the last phase of our tests, we formally compared the trade-off of all twenty-setups with the aim to identify and eliminate setups that are both slower and produce worse solutions than other setups, and are therefore dominated by other setups. To identify dominating and dominated setups, we calculated the average savings per benchmark set obtained by the setups over all the initial solutions as well as their average execution times. Tests were also performed on the smaller *mval-IF-3L* instances. A setup was then flagged as dominated on a benchmark set if another setup produced the same or better quality local optima, but required less computational time to do so.

The domination of the setups in terms of average cost savings and computational time is shown in Figure 13. We again refer the reader to Table 4 for a full description of the setups. The Accelerated-Extended-Greedy (AEG-f) setups at the four different f levels performed well and were part of the dominating setups on all four benchmark sets. Accelerated-Reduced-Greedy-0.25 (ARG-0.25) was the quickest setup and was therefore also part of the dominating setups, but its average improvement was low, being close to or less than 1% on the waste collection sets and close to 2.5% on *mval-IF-3L*. The ARG-f and Accelerated-Extended-Greedy-f (AEG-f) setups produced low improvements over the initial solutions at low f values. The savings increased with higher f levels, and without significantly increasing computational times. This shows that the *Static-Move-Descriptors* and *Greedy-Compound-Independent-Moves* mechanisms, on their own, are effective, particularly when linked with the extended move operators. On the larger *Cen-IF* and *Lpr-IF* sets, the Basic-Full-Best (BFB) and Basic-Reduced-Best (BRB) setups had the highest average cost savings, but their execution times were very long. Extending the setups to or using them within meta-

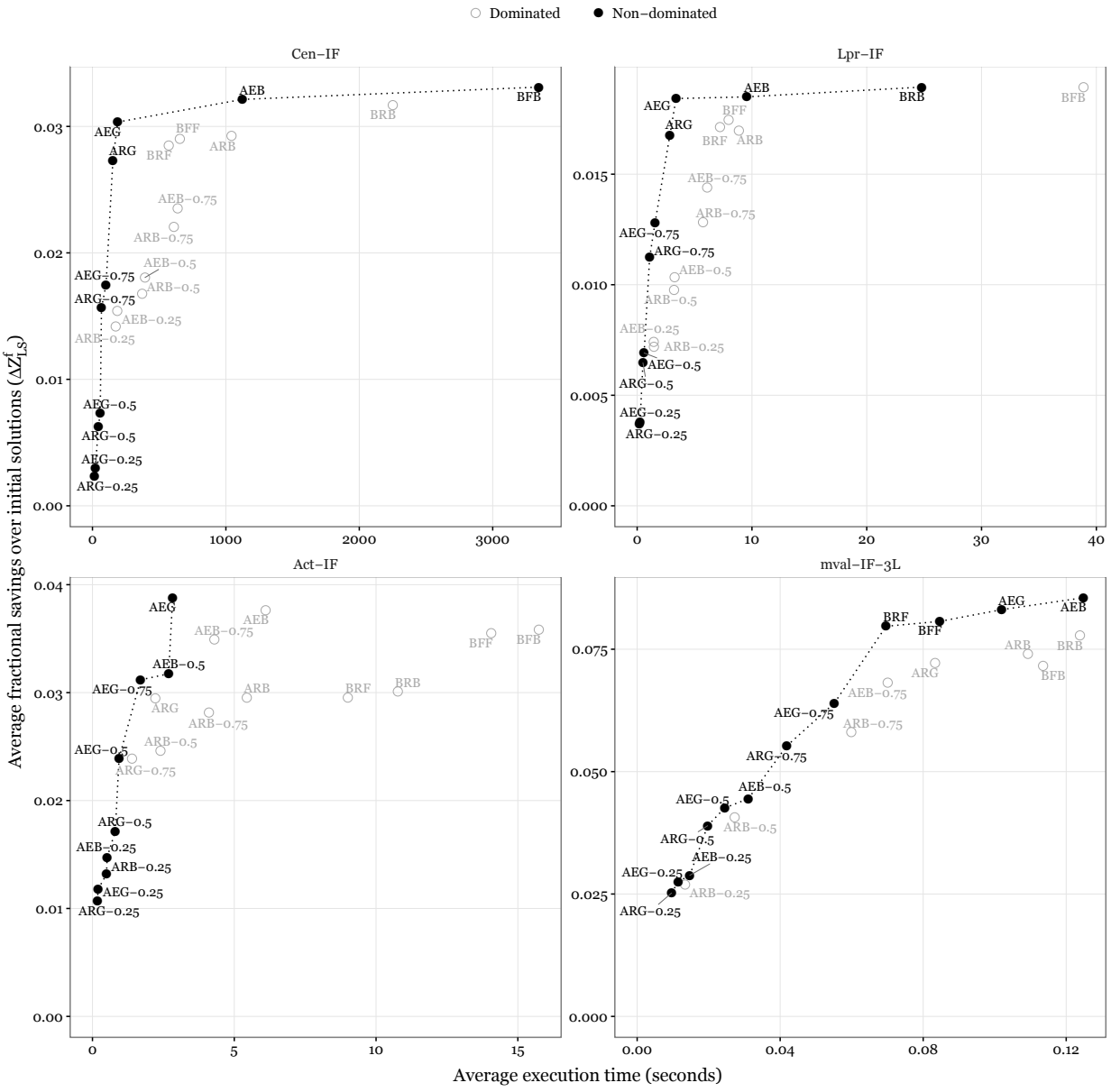


Figure 13: Dominated and non-dominated local search setups in terms of fractional cost savings and execution time of the setups on four MCARP-TIF benchmark sets. A full list of acronyms used in the setup can be found in Table 4.

heuristics will only prolong total execution times, and we, therefore, recommend against using them when dealing with realistically sized instances.

The performance of the setups was different on the *mval-IF-3L* instances compared to the waste collection sets. Significant cost savings were obtained, in excess of 7.5%, for some of the setups, and the time to do so was quite modest, being at most 0.12 seconds for even the slowest setup. It is also the only set on which the BRF and BFF setups were part of the dominating setups. With its low computational time, BRF and BFF would be good candidates to use for metaheuristics, if the tests were limited to *mval-IF-3L* instance. On the waste collection sets, both were always dominated by AEG. Tests on the waste collection instances show that BRF and BFF cannot be used for waste collection planning. This confirms the need for computational tests to be performed on realistic waste collection instances, as the performance of heuristics on small instances cannot be used to predict their performance in more practical settings.

Modified versions of the sixteen LS setups were further tested on the classical *Lpr* and *mval* MCARP instances. All these setups, which we refer to as MCARP setups, were modified so as to not consider changes in IF positions and to ignore the route duration limit, L, since they are not applicable to the MCARP. Other than that, the MCARP setups were identical to the ones tested on the MCARPTIF. This automatically improves the efficiency of the setups compared to the MCARPTIF versions. Results for the modified MCARP setups are shown shown in Figure 14. Compared to results on the *Lpr-IF* and *mval-IF-3L* instances, the improvement on the MCARP instances fell within

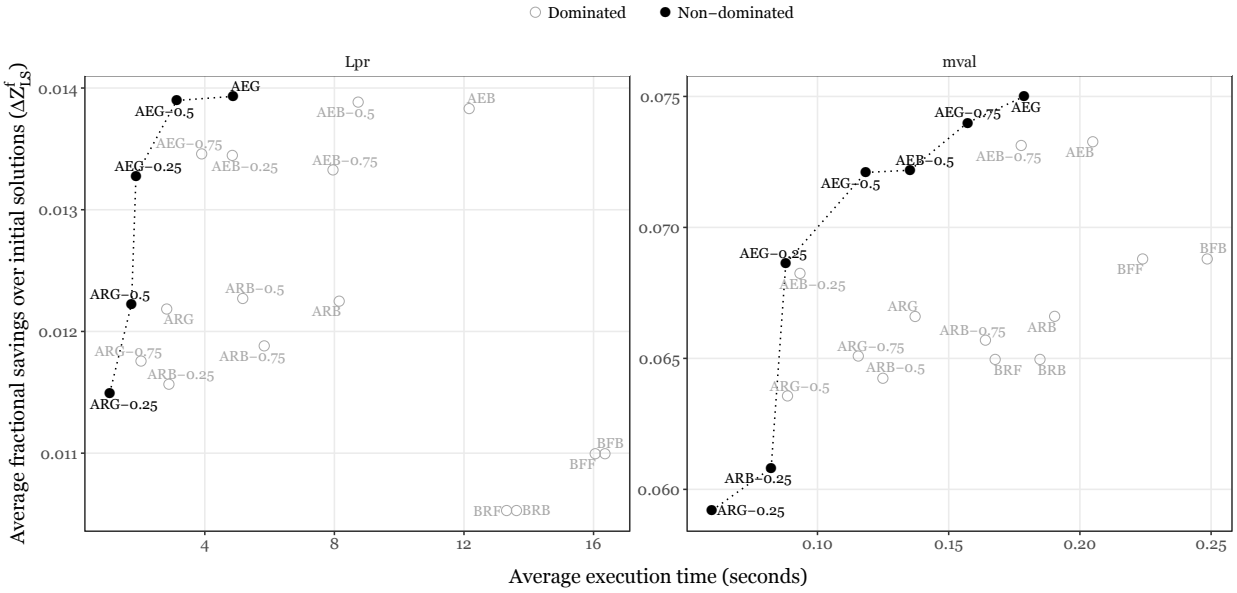


Figure 14: Dominated and non-dominated local search setups in terms of fractional cost savings and execution time of the setups on two MCARP benchmark sets. A full list of acronyms used in the setup can be found in Table 4.

a much smaller range of between 0.011 and 0.014 on *Lpr* and between 0.055 and 0.075 on *mval*. The best achievable savings were also less on the MCARP instances. This may be attributed to the initial solutions on the MCARPTIF being worse than the MCARP, thus leaving more room for improvement. As expected, the setups were generally quicker on *Lpr* compared to *Lpr-IF* since moves involving IFs do not have to be considered, and the route duration limit is never checked. The setups were actually slower on the *mval* instances, compared to *mval-IF-3L*, but still had average execution times of below 0.25 seconds. On both sets, the AEG-f setups were part of the dominating setups, except for AEG-0.75 on *Lpr* where it was dominated by AEG-0.5. ARG-0.25 was always the quickest but produced low improvements. On the MCARP instances, the other accelerated setups featured less prominently, and the classical setups did not feature in the dominating setups at all. This demonstrates the robustness of AEG-f in that it performed consistently well on the MCARP and MCARPTIF.

AEG-f, which combines *Static-Move-Descriptors*, the extended move operators, and the *Greedily-Compounding-Independent-Moves* strategy, but without *Nearest-Neighbour-Lists* performed the best. Based on the results, we recommend that AEG be used either directly or within metaheuristic algorithms when dealing with large waste collection instances. It can also be used as a starting point to develop better LS versions for the MCAR and MCARPTIF by implementing additional move operators within the acceleration mechanisms. Lastly, if required, the execution time of AEG can be decreased by activating the *Nearest-Neighbour-Lists* and by decreasing f. It is therefore not required to switch to a different setup, such as ARG-0.25, if the available execution time is limited.

7. Conclusion

Efficient LS setups is an important area of research for CARPs, given their use within metaheuristic applications which currently struggle to deal with realistically sized instances. In this paper, we extend our previous work on greedy constructive heuristics for the MCARPTIF by developing efficient Local Search improvement heuristics for the problem. Three acceleration mechanisms were developed and linked with LS, of which the setup with *Static-Move-Descriptors* and *Greedy-Compound-Independent-Moves* performed the best. The third acceleration mechanism, *Nearest-Neighbour-Lists*, had the desired effect of improving the efficiency of LS, but its resulting reduction in solution quality limited its application. The long execution times of LS linked only with *Static-Mode-Descriptors* also limited its application on realistically sized instances.

On instances with more than 1000 required arcs and edges, the basic LS setups took between fifteen minutes and three hours to improve a single solution. The accelerated setups took at most four minutes to improve the same solutions, with the most efficient version taking less than 60 seconds. Our accelerated LS implementations were thus effective in improving the initial solutions of constructive heuristics, and efficient enough to do so within short execution time-limits. The significance of our research contribution on LS heuristics extends beyond the MCARPTIF. Our acceleration mechanisms can be applied as-is to LS for the CARP and MCARP, thereby improving the efficiency of metaheuristics that rely on LS and allowing them to more effectively deal with large instances.

In addition to our proposed future research on metaheuristics, there exist much scope to improve the methods presented in this paper. To conclude the paper, we briefly discuss some of these opportunities. The implementations of *Static-Move-Descriptors* and *Greedy-Compound-Independent-Moves* can be improved by using priority-queues instead of sorting the move list at each iteration. More intelligent applications of *Nearest-Neighbour-Lists* may also improve its performance, for example, by applying the mechanism only to specific move operators. The number of nearest neighbours can also be parameterised for each move operator. The move neighbourhood can be extended by considering consecutive task relocations and exchanges, and by using more advanced versions of *two-opt*. The sequential search techniques, developed in Irnich et al. [14] for the VRP, can then be adapted to MCARPTIF LS heuristics to scan the extended neighbourhood efficiently. It may also be worth directly incorporating the acceleration mechanisms tested in this paper to further enhance sequential search. Recently, Vidal [22] introduced a single neighbourhood extension through structural neighbourhood decomposition whereby the unique subproblem of determining the optimal service orientation of edge-tasks can be efficiently determined for each LS move. Together with *Nearest-Neighbour-Lists*, the author successfully applied partial move-cost lower-bounds to limit the number of LS moves to evaluate, thereby improving its efficiency. Here too, an opportunity exists to combine the structural neighbourhood decomposition and partial move-cost lower-bounds with our acceleration mechanisms to further improve the performance of LS.

References

[1] Belenguer, J., Benavent, E., Lacomme, P., and Prins, C. (2006). Lower and upper bounds for the mixed capacitated arc routing problem. *Computers & Operations Research*, 33(12):3363–3383.

[2] Beullens, P., Muyldermans, L., Cattrysse, D., and Van Oudheusden, D. (2003). A guided local search heuristic for the capacitated arc routing problem. *European Journal of Operational Research*, 147(3):629–643.

[3] Chen, Y. and Hao, J.-K. (2017). Two phased hybrid local search for the periodic capacitated arc routing problem. *European Journal of Operational Research*, 264(1):55–65.

[4] Chen, Y., Hao, J.-K., and Glover, F. (2016). A hybrid metaheuristic approach for the capacitated arc routing problem. *European Journal of Operational Research*, 253(1):25–39.

[5] Corberán, Á. and Laporte, G. (2015). *Arc routing: problems, methods, and applications*, volume 20. SIAM.

[6] Corberán, A. and Prins, C. (2010). Recent results on arc routing problems: an annotated bibliography. *Networks*, 56(1):50–69.
[7] Coutinho-Rodrigues, J., Rodrigues, N., and Clímaco, J. (1993). Solving an urban routing problem using heuristics: a successful case study. *Belgian Journal of Operations Research, Statistics and Computer Science*, 33(1):2.
[8] Dell'Amico, M., Díaz, J. C. D., Hasle, G., and Iori, M. (2016). An adaptive iterated local search for the mixed capacitated general routing problem. *Transportation Science*, 50(4):1223–1238.
[9] Dror, M., editor (2000). *Arc Routing: Theory, Solutions, and Applications*. Boston: Kluwer Academic Publishers.
[10] 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.
[11] Ghiani, G., Guerriero, F., Laporte, G., and Musmanno, R. (2004). Tabu search heuristics for the arc routing problem with intermediate facilities under capacity and length restrictions. *Journal of Mathematical Modelling and Algorithms*, 3(3):209–223.
[12] Golden, B. L. and Wong, R. T. (1981). Capacitated arc routing problems. *Networks*, 11(3):305–315.
[13] Gouveia, L., Mourão, M. C., and Pinto, L. S. (2010). Lower bounds for the mixed capacitated arc routing problem. *Computers & Operations Research*, 37(4):692–699.
[14] Irnich, S., Funke, B., and Grünert, T. (2006). Sequential search and its application to vehicle-routing problems. *Computers & Operations Research*, 33(8):2405–2429.
[15] Lacomme, P., Prins, C., and Ramdane-Chérif, W. (2004). Competitive memetic algorithms for arc routing problems. *Annals of Operations Research*, 131(4):159–185.
[16] Lenstra, J. K. (2003). *Local search in combinatorial optimization*. Princeton University Press.
[17] Mourão, M. C. and Pinto, L. S. (2017). An updated annotated bibliography on arc routing problems. *Networks*, in press.
[18] Muyldermans, L. and Pang, G. (2015). Variants of the capacitated arc routing problem. In Corberán, Á. and Laporte, G., editors, *Arc routing: problems, methods, and applications*, volume 20 of *MOS-SIAM Series on Optimization*, chapter 10, pages 223–253. SIAM.
[19] Prins, C. (2015). The capacitate arc routing problem: heuristics. In Corberán, Á. and Laporte, G., editors, *Arc routing: problems, methods, and applications*, volume 20 of *MOS-SIAM Series on Optimization*, chapter 7, pages 131–157. SIAM.
[20] Santos, L., Coutinho-Rodrigues, J., and Current, J. R. (2010). An improved ant colony optimization based algorithm for the capacitated arc routing problem. *Transportation Research Part B: Methodological*, 44(2):246–266.
[21] Talbi, E. (2009). *Metaheuristics: From design to implementation*. Wiley, New Jersey.
[22] Vidal, T. (2017). Node, edge, arc routing and turn penalties: Multiple problems—one neighborhood extension. *Operations Research*, 65(4):992–1010.
[23] Willemse, E. J. (2016). *Heuristics for large-scale Capacitated Arc Routing Problems on mixed networks*. PhD thesis, University of Pretoria, Pretoria. Available online from `http://hdl.handle.net/2263/57510` (Last viewed on 2017-01-16).
[24] Willemse, E. J. and Joubert, J. W. (2016a). Benchmark dataset for undirected and mixed capacitated arc routing problems under time restrictions with intermediate facilities. *Data in Brief*, 8:972–977.
[25] Willemse, E. J. and Joubert, J. W. (2016b). Constructive heuristics for the mixed capacity arc routing problem under time restrictions with intermediate facilities. *Computers & Operations Research*, 68:30–62.
[26] Willemse, E. J. and Joubert, J. W. (2016c). Library of benchmark test sets for variants of the capacitated arc routing problem under time restrictions with intermediate facilities, v3. *Mendeley Data*. Available online from `http://dx.doi.org/10.17632/9x4vd92rcj.3`.
[27] Willemse, E. J. and Joubert, J. W. (2016d). Splitting procedures for the mixed capacitated arc routing problem under time restrictions with intermediate facilities. *Operations Research Letters*, 44(5):569–574.
[28] Zachariadis, E. E. and Kiranoudis, C. T. (2010). A strategy for reducing the computational complexity of local search-based methods for the vehicle routing problem. *Computers & Operations Research*, 37(12):2089–2105.

Appendix A. Algorithm descriptions

Algorithms 2 and 3 give the *Nearest-Neighbour-List* implementations for *exchange* and *cross* move operators. The implementation for *relocate* can be found in Algorithm 1 in Section 5.1. For *Greedy-Compound-Independent-Moves*, Algorithm 4 is used to make all independent moves whose move independence is checked via Algorithm 5. For *Static-Move-Descriptors*, the savings list can be updated using Algorithm 6 that incorporates the *Nearest-Neighbour-Lists* algorithms. All three acceleration mechanisms are then be combined as shown in Algorithm 7.

Algorithm 2: *Find-Exchange-Moves*

Input : Current solution $\boldsymbol{T}$; savings threshold, $\Delta\overline{Z}$; tasks $\boldsymbol{R}$ to be considered for exchange; savings list $\boldsymbol{M}$ consisting of information needed to implement moves.
Output: Updated savings list $\boldsymbol{M}$, with information of moves with savings less (better) than $\Delta\overline{Z}$ added to $\boldsymbol{M}$.

for $u \in \boldsymbol{R}$ **do**
  $(i, j, k) = T^{-1}(u)$;
  $u_{\text{pre}} = T_{i,j,k-1}$;
  $s = \lceil f \times |\boldsymbol{R}| \rceil$;
  $\boldsymbol{R}' = \boldsymbol{R} \cap \{N_{u_{\text{pre}},1}, \ldots, N_{u_{\text{pre}},s}\}$;
  for $v \in \boldsymbol{R}'$ **do**
    $(l, m, n) = T^{-1}(v)$;
    if $u < v$ **then**
      // an *exchange* between v and u is the same as an *exchange* between u and v, so only one has to be evaluated //;
      Using $\boldsymbol{T}$ and i, j, k of u, and l, m, n of v, calculate ΔZ for the *exchange* move;
      if $\Delta Z < \Delta\overline{Z}$ **then**
        $\boldsymbol{M} = \boldsymbol{M} \cup \{(\Delta Z, move_i = 2, u, v)\}$;
return $(\boldsymbol{M})$

Algorithm 3: *Find-Cross-Moves*

Input : Current solution $\boldsymbol{T}$; savings threshold, $\Delta\overline{Z}$; tasks $\boldsymbol{R}_T$ to be considered for the cross move; savings list $\boldsymbol{M}$ consisting of information needed to implement moves.
Output: Updated savings list $\boldsymbol{M}$, with information of moves with savings less (better) than $\Delta\overline{Z}$ added to $\boldsymbol{M}$.

for $u \in \boldsymbol{R}_T$ **do**
  $(i, j, k) = T^{-1}(u)$;
  $u_{\text{pre}} = T_{i,j,k-1}$;
  $s = \lceil f \times |\boldsymbol{R}| \rceil$;
  $\boldsymbol{R}'_v = \boldsymbol{R}_v \cap \{N_{u_{\text{pre}},1}, \ldots, N_{u_{\text{pre}},s}\}$;
  for $v \in \boldsymbol{R}'_v$ **do**
    $(l, m, n) = T^{-1}(v)$;
    if $u < v$ **then**
      Using $\boldsymbol{T}$ and i, j, k of u, and l, m, n of v, calculate ΔZ for the *cross* move;
      if $\Delta Z < \Delta\overline{Z}$ **then**
        $\boldsymbol{M} = \boldsymbol{M} \cup \{(\Delta Z, move_i = 3, u, v)\}$;
return $(\boldsymbol{M})$

Algorithm 4: *Greedy-Compound-Moves*

Input : Current solution, $\boldsymbol{T}$; savings threshold, $\Delta\overline{Z} = 0$; savings-list, $\boldsymbol{M}$.
Output: Neighbouring solution, $\boldsymbol{T}'$, with independent moves implemented on $\boldsymbol{T}$; total savings, ΔZ_{total}, resulting from the compounded moves; dependent task set $\boldsymbol{C}$ for cost-link changes.

$\boldsymbol{C} = \varnothing$;
$\boldsymbol{T}' = \boldsymbol{T}$;
Order $\boldsymbol{M}$ from the best to worst improving move;
for $\pi \in \boldsymbol{M}$ **do**
  $(\Delta Z, move_i, u, v) = \pi$;
  if $\Delta Z < \Delta\overline{Z}$ **then**
    if $u, inv(u), v, inv(v) \notin \boldsymbol{C}$ *and the move is feasible* **then**
      $\boldsymbol{C}$ =*Update-Move-Dependence-Task-Sets*$(\pi, \boldsymbol{C})$ // Algorithm 5 //;
      Implement the move on $\boldsymbol{T}'$;
      $\Delta Z_{\text{total}} = \Delta Z_{\text{total}} + \Delta Z$;
      // for a pure *find-best* implementation the heuristic would stop here and immediately return $\boldsymbol{T}$ and ΔZ_{total} //;
return $(\boldsymbol{T}', \Delta Z_{\text{total}}, \boldsymbol{U}_a, \boldsymbol{U}_b)$

Algorithm 5: *Update-Move-Dependence-Task-Sets*

Input : Move information, π; dependent task set, $\boldsymbol{C}$.
Output: Updated set, $\boldsymbol{C}'$ with tasks from cost-links of move π added to the sets.

$(\Delta Z, move_i, u, v,) = \pi$;
if $move_i = 1$ **then** // if it's a *relocate* move //
  $\boldsymbol{R} = \{pre(u), u, post(u), pre(v), v\}$;
if $move_i = 2$ **then** // if it's an *exchange* move //
  $\boldsymbol{R} = \{pre(u), u, post(u), pre(v), v, post(v)\}$;
if $move_i = 3$ **then** // if it's a *cross* move //
  $\boldsymbol{R} = \{pre(u), u, post(u), pre(v), v, post(v)\}$;
$\boldsymbol{R}' \cup \{inv(u') : u' \in \boldsymbol{R} \text{ and } inv(u') \neq 0\}$;
$\boldsymbol{C}' = \boldsymbol{C} \cup \boldsymbol{R}'$;
return $(\boldsymbol{C}')$

Algorithm 6: *Update-Savings-List*

Input : Dependent task set, $\boldsymbol{C}$, cost-link task changes; move savings list, $\boldsymbol{M}$; move-cost threshold, $\Delta\overline{Z} = 0$.
Output: Updated move savings list, $\boldsymbol{M}'$

$\boldsymbol{M}' = \varnothing$;
for $\pi \in \boldsymbol{M}$ **do**
  $(\Delta Z, move_i, u, v) = \pi$ **if** $u, inv(u), v, inv(v) \in \boldsymbol{C}$ **then**
    $\boldsymbol{M}' = \boldsymbol{M}' \cup \{\pi\}$
$\boldsymbol{M}' =$ *Find-Relocate-Moves*$(\Delta\overline{Z}, \boldsymbol{C}, \boldsymbol{R}_T, \boldsymbol{M}')$ // Algorithm 1 //;
$\boldsymbol{M}' =$ *Find-Relocate-Moves*$(\Delta\overline{Z}, \boldsymbol{R}/\boldsymbol{C}, \boldsymbol{U}_b, \boldsymbol{M}')$;
$\boldsymbol{M}' =$ *Find-Exchange-Moves*$(\Delta\overline{Z}, \boldsymbol{C}, \boldsymbol{R})$ // Algorithm 2 //;
$\boldsymbol{M}' =$ *Find-Exchange-Moves*$(\Delta\overline{Z}, \boldsymbol{R}/\boldsymbol{C}, \boldsymbol{C}, \boldsymbol{M}')$;
$\boldsymbol{M}' =$ *Find-Cross-Moves*$(\Delta\overline{Z}, \boldsymbol{C}, \boldsymbol{R}_T)$ // Algorithm 3 //;
$\boldsymbol{M}' =$ *Find-Cross-Moves*$(\Delta\overline{Z}, \boldsymbol{R}_T/\boldsymbol{C}, \boldsymbol{C}, \boldsymbol{M}')$;
return $(\boldsymbol{M}')$

Algorithm 7: *Accelerated-Local-Search*

Input : Initial solution, $\boldsymbol{T}^{(0)} \in \boldsymbol{X}$, savings threshold $\Delta\overline{Z} = 0$.
Output: Local optimum solution, $\boldsymbol{T}^{(t)}$

$t = 0$;
$\boldsymbol{M} = \varnothing$;
$\boldsymbol{M} =$ *Find-Relocate-Moves*$(\boldsymbol{T}^{(0)}, \Delta\overline{Z}, \boldsymbol{R}, \boldsymbol{R}_T, \boldsymbol{M})$ // Algorithm 1 //;
$\boldsymbol{M} =$ *Find-Exchange-Moves*$(\boldsymbol{T}^{(0)}, \Delta\overline{Z}, \boldsymbol{R}, \boldsymbol{R}, \boldsymbol{M})$ // Algorithm 2 //;
$\boldsymbol{M} =$ *Find-Cross-Moves*$(\boldsymbol{T}^{(0)}, \Delta\overline{Z}, \boldsymbol{R}_T, \boldsymbol{R}_T, \boldsymbol{M})$ // Algorithm 3 //;
repeat
  if $\boldsymbol{M} \neq \varnothing$ **then**
    $(\boldsymbol{T}', \Delta Z_{\text{total}}, \boldsymbol{C}) =$ *Greedy-Compound-Moves*$(\boldsymbol{T}^{(t)}, \Delta\overline{Z}, \boldsymbol{M})$ // Algorithm 4 //;
    if $\Delta Z_{total} < \Delta\overline{Z}$ **then**
      Set $\boldsymbol{T}^{(t+1)} = \boldsymbol{T}'$;
      $\boldsymbol{M}'' =$ *Update-Savings-List*$(\boldsymbol{C}, \boldsymbol{M}, \Delta\overline{Z})$ // Algorithm 6 //;
      $\boldsymbol{M} = \boldsymbol{M}''$;
      $t = t + 1$;
    else a feasible move could not be found;
  else an improving move could not be found;
until a feasible improving move could not be found;
return $(\boldsymbol{T}^{(t)})$

Appendix B. Detailed results tables

Table B.1 shows the performance of our best LS setup, Accelerated-Extended-Greedy (AEG) without *Nearest-Neighbour-Lists* on the four MCARPTIF benchmark sets. The total reported execution time, t includes the time to construct the initial solution and for AEG to improve it to a local optimum. AEG was paired with three different constructive heuristics from Willemse and Joubert (2016b). *Efficient-Route-Cluster* (ERC) and *Path-Scanning* (PS) generates five different solutions, of which only the best is returned and improved using AEG. *Improved-Merge (IM)* returns a single solution that is improved with AEG.

Tables B.2 and B.3 show the performance AEG against the Memetic-Algorithm of Belenguer et al. (2006) on the *Lpr* and *mval* MCARP benchmark sets. Lower-bounds for the instances were taken from Belenguer et al. (2006) and Gouveia et al. (2010). The lower-bound gaps were calculated as:

$$\text{Gap} = \frac{Z - LB}{Z}. \tag{B.1}$$

Full results tables on all the MCARP and MCARPTIF instances, with the execution time and final cost of each LS setup per problem instance and initial solution, can be found in the online supplementary material.

Table B1
Solution cost and execution time, in seconds, of Accelerated-Extended-Greedy (AEG) linked with three different constructive heuristics on the MCARPTIF benchmark sets.

	Solution cost (Z)			Execution time (s)		
Instance	ERC-AEG	IM-AEG	PS-AEG	ERC-AEG	IM-AEG	PS-AEG
Act-IF-a	22353	22519	22277	1.57	1.7	2.77
Act-IF-b	72015	73238	72002	4.21	6.66	6.14
Act-IF-c	49984	50297	49754	3.18	3.96	2.91
Cen-IF-a	234111	232868	240313	1.85	2.25	1.53
Cen-IF-b	584514	592205	586983	3.72	3.64	3.51
Cen-IF-c	521143	514852	527537	3.37	3.03	3.51
Lpr-IF-a-01	13609	13686	13589	1.01	0.58	1.31
Lpr-IF-a-02	28377	28346	28635	0.42	0.66	0.64
Lpr-IF-a-03	77988	78159	78154	1.46	2.12	1.61
Lpr-IF-a-04	131818	133109	131185	1.33	1.88	3.4
Lpr-IF-a-05	208950	210057	208111	3.17	3.54	2.46
Lpr-IF-b-01	14876	14870	14875	0.6	0.76	0.38
Lpr-IF-b-02	28937	28840	29309	2.97	1.61	0.66
Lpr-IF-b-03	79511	79438	79433	1.32	1.04	2.01
Lpr-IF-b-04	131254	133147	130976	1.4	1.35	0.58
Lpr-IF-b-05	217547	220831	217626	1.53	2.23	1.86
Lpr-IF-c-01	18773	18866	18803	1.41	0.61	1.08
Lpr-IF-c-02	36644	36727	36510	0.77	0.74	0.5
Lpr-IF-c-03	113532	113583	112847	0.76	2.71	0.8
Lpr-IF-c-04	172185	172905	173729	2.12	2.77	2.83
Lpr-IF-c-05	271529	271048	270376	2.77	2.58	2.57
mval-IF-3L-1A	250	278	277	1.63	1.15	0.42
mval-IF-3L-1B	325	321	328	0.9	0.39	1.09
mval-IF-3L-1C	380	401	387	1.39	0.67	1.12
mval-IF-3L-2A	501	413	423	0.45	0.52	1.32
mval-IF-3L-2B	435	485	437	0.29	1	0.39
mval-IF-3L-2C	535	565	488	0.98	2.19	1.44
mval-IF-3L-3A	141	143	142	0.63	0.49	0.67
mval-IF-3L-3B	158	169	172	1.68	0.69	0.57
mval-IF-3L-3C	138	142	133	0.29	0.55	0.46
mval-IF-3L-4A	653	682	672	0.8	0.26	0.57
mval-IF-3L-4B	758	792	761	1.36	0.93	1.17
mval-IF-3L-4C	798	786	782	1.07	0.57	1.2
mval-IF-3L-4D	852	864	789	1.48	0.49	1.1
mval-IF-3L-5A	799	779	811	0.39	0.24	0.61
mval-IF-3L-5B	737	741	775	0.48	0.56	0.64
mval-IF-3L-5C	821	890	849	0.92	1.45	1.06
mval-IF-3L-5D	894	888	841	1.13	1.73	1.69
mval-IF-3L-6A	360	365	372	0.71	0.8	0.68
mval-IF-3L-6B	359	412	371	1.09	0.7	0.77
mval-IF-3L-6C	477	453	493	1.2	2.07	1.14
mval-IF-3L-7A	396	408	430	0.78	0.6	0.49
mval-IF-3L-7B	462	491	485	1.13	1.27	1
mval-IF-3L-7C	517	532	541	0.47	0.65	1.06
mval-IF-3L-8A	684	670	650	1.26	0.35	1.47
mval-IF-3L-8B	661	635	600	0.51	1.17	0.79
mval-IF-3L-8C	689	684	690	2.19	1.23	1.26
mval-IF-3L-9A	611	583	580	1.17	0.75	0.86
mval-IF-3L-9B	532	551	541	0.84	0.78	0.92
mval-IF-3L-9C	562	562	564	0.83	0.76	0.78
mval-IF-3L-9D	647	659	677	0.41	1.02	0.26
mval-IF-3L-10A	841	844	837	0.49	1.04	0.42
mval-IF-3L-10B	820	835	815	0.8	0.41	1.18
mval-IF-3L-10C	761	770	776	1.24	1.24	0.52
mval-IF-3L-10D	810	829	813	0.51	1.05	0.58

Table B2
Comparison between the Memetic Algorithm (MA) in Belenguer et al. (2006) and the Accelerated-Extended-Greedy local search setup with three different constructive heuristics on the *Lpr* MCARP benchmark set.

		MA			ERC-AEG			IM-AEG			PS-AEG		
Instance	LB	*Z*	Gap (%)	*t* (s)	*Z*	Gap (%)	*t* (s)	*Z*	Gap (%)	*t* (s)	*Z*	Gap (%)	*t* (s)
a-01	13484	13484	0	1.27	13512	0.21	0.05	13609	0.93	0.04	13484	0.00	0.04
a-02	28052	28052	0	1.48	28232	0.64	0.26	28237	0.66	0.21	28236	0.66	0.29
a-03	76115	76155	0.05	1824.25	77321	1.58	1.49	76986	1.14	3.17	77576	1.92	1.92
a-04	126946	127930	0.78	3600	130053	2.45	7.12	128330	1.09	7.33	129610	2.10	5.26
a-05	202736	206086	1.65	3600	209322	3.25	17.54	207550	2.37	16.18	207865	2.53	25.88
b-01	14835	14835	0	0.06	14908	0.49	0.03	14918	0.56	0.06	14876	0.28	0.04
b-02	28654	28654	0	1.14	28758	0.36	0.14	29054	1.40	0.20	28733	0.28	0.14
b-03	77859	77878	0.02	1654.17	79222	1.75	1.01	79498	2.11	2.20	79204	1.73	0.88
b-04	126932	127454	0.41	3600	130576	2.87	3.34	128856	1.52	6.09	129437	1.97	4.14
b-05	209791	212279	1.19	3600	215170	2.56	11.81	213974	1.99	21.16	214311	2.15	15.08
c-01	18639	18639	0	0.31	18736	0.52	0.04	18780	0.76	0.09	18779	0.75	0.06
c-02	36339	36339	0	10.05	36646	0.84	0.17	36624	0.78	0.40	36777	1.21	0.09
c-03	111117	111632	0.46	3600	113870	2.48	3.55	112483	1.23	5.36	113062	1.75	2.82
c-04	168441	169487	0.62	3600	172389	2.34	7.07	171169	1.62	20.50	171660	1.91	7.67
c-05	257890	260538	1.03	3600	263643	2.23	20.11	260977	1.20	48.26	262955	1.96	20.83
Mean			*0.41*	*1912.85*		*1.64*	*4.92*		*1.29*	*8.75*		*1.41*	*5.68*

Notes: Lower-bound (LB); solution cost (*Z*); percentage gap from LB (Gap); total execution time, in seconds, of the solution technique (*t*).

Table B3
Comparison between the Memetic Algorithm (MA) in Belenguer et al. (2006) and the Accelerated-Extended-Greedy local search setup with three different constructive heuristics on the *mval* MCARP benchmark set.

		MA			ERC-AEG			IM-AEG			PS-AEG		
Instance	LB	*Z*	Gap (%)	*t* (s)	*Z*	Gap (%)	*t* (s)	*Z*	Gap (%)	*t* (s)	*Z*	Gap (%)	*t* (s)
1A	230	230	0	0.1	254	10.43	0.05	247	7.39	0.14	239	3.91	0.05
1B	261	261	0	0.27	306	17.24	0.06	299	14.56	0.16	296	13.41	0.15
1C	309	315	1.94	34.08	337	9.06	0.11	347	12.30	0.14	386	24.92	0.19
2A	324	324	0	0.2	397	22.53	0.04	341	5.25	0.05	348	7.41	0.09
2B	395	395	0	0.31	413	4.56	0.05	407	3.04	0.06	413	4.56	0.06
2C	521	526	0.96	31.57	581	11.52	0.15	560	7.49	0.07	578	10.94	0.21
3A	115	115	0	0.45	126	9.57	0.08	127	10.43	0.07	134	16.52	0.06
3B	142	142	0	53.36	146	2.82	0.07	149	4.93	0.08	148	4.23	0.05
3C	166	166	0	6.65	187	12.65	0.11	170	2.41	0.07	182	9.64	0.05
4A	580	580	0	17.82	678	16.90	0.15	618	6.55	0.24	645	11.21	0.17
4B	650	650	0	2.23	695	6.92	0.23	702	8.00	0.22	701	7.85	0.26
4C	630	631	0.16	147.3	754	19.68	0.20	663	5.24	0.36	707	12.22	0.42
4D	746	776	4.02	134.25	852	14.21	0.40	815	9.25	0.32	823	10.32	0.52
5A	597	597	0	19.4	657	10.05	0.17	647	8.38	0.32	644	7.87	0.19
5B	613	615	0.33	141.14	737	20.23	0.02	657	7.18	0.20	651	6.20	0.22
5C	697	697	0	81.07	775	11.19	0.21	723	3.73	0.27	736	5.60	0.19
5D	719	757	5.29	113.3	849	18.08	0.17	808	12.38	0.46	961	33.66	0.02
6A	326	326	0	11.65	347	6.44	0.09	353	8.28	0.19	356	9.20	0.11
6B	317	317	0	15.95	378	19.24	0.07	327	3.15	0.13	353	11.36	0.17
6C	365	375	2.74	58.18	434	18.90	0.15	404	10.68	0.17	528	44.66	0.02
7A	364	364	0	0.75	407	11.81	0.13	382	4.95	0.34	395	8.52	0.08
7B	412	412	0	7.47	465	12.86	0.19	469	13.83	0.20	475	15.29	0.25
7C	424	428	0.94	113.4	526	24.06	0.28	468	10.38	0.19	486	14.62	0.29
8A	581	581	0	76.78	644	10.84	0.16	628	8.09	0.34	630	8.43	0.23
8B	531	531	0	6.02	595	12.05	0.28	556	4.71	0.18	603	13.56	0.12
8C	617	638	3.4	77.84	749	21.39	0.25	681	10.37	0.19	726	17.67	0.58
9A	458	458	0	24	510	11.35	0.31	492	7.42	0.48	520	13.54	0.24
9B	453	453	0	17.95	496	9.49	0.37	495	9.27	0.57	496	9.49	0.28
9C	428	434	1.4	246.28	488	14.02	0.26	486	13.55	0.49	484	13.08	0.29
9D	514	520	1.17	253.22	623	21.21	0.51	567	10.31	0.52	568	10.51	0.53
10A	634	634	0	128.6	708	11.67	0.27	689	8.68	0.52	675	6.47	0.40
10B	661	662	0.15	313.7	714	8.02	0.43	735	11.20	0.58	703	6.35	0.32
10C	623	624	0.16	314.05	764	22.63	0.04	654	4.98	0.49	681	9.31	0.44
10D	643	650	1.09	267.05	729	13.37	0.56	714	11.04	0.26	715	11.20	0.49
Mean			*0.70*	*79.89*		*13.74*	*0.19*		*8.22*	*0.27*		*12.17*	*0.23*

Notes: Lower-Bound (LB); Solution cost (*Z*); Percentage gap from LB (Gap); total execution time, in seconds, of the solution technique (*t*).