UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

Heuristics for large-scale Capacitated Arc Routing Problems on mixed networks

by

Elias Jakobus Willemse
u04405013

A thesis submitted in fulfilment of the requirements for the degree

Philosophiae Doctor (Industrial Engineering)

in the

Faculty of Engineering, Built Environment and Information Technology

University of Pretoria
Pretoria, South Africa

July 2016

Abstract

Heuristics for large-scale Capacitated Arc Routing Problems on mixed networks

by

Elias Jakobus Willemse

Supervisor	:	Prof. Johan W. Joubert
Department	:	Industrial and Systems Engineering
University	:	University of Pretoria
Degree	:	Philosophiae Doctor (Industrial Engineering)
Thesis examined by	:	Prof. Richard Eglese (Lancaster University) Prof. Luc Muyldermans (Nottingham University) Dr. Jacomine Grobler (University of Pretoria)

Residential waste collection is an expensive activity performed daily on large metropolitan areas throughout the year. Even a small improvement in waste collection and transfer operations can therefore lead to significant cost savings. A promising improvement area is to design better waste collection routes.

In this thesis we show that the problem of designing optimal collection routes for waste collection vehicles can be modelled as the Mixed Capacitated Arc Routing Problem under Time Restrictions with Intermediate Facilities (MCARPTIF). The problem is a generalisation of the Capacitated Arc Routing Problem (CARP) and takes into consideration a mixed road network consisting of one- and two-way streets; a vehicle capacity that limits the amount of waste a vehicle can carry at any given time; Intermediate Facilities (IFs) where vehicles are allowed to unload their waste and resume their collection rounds; and a time restriction which prohibits the duration of a route from exceeding a time-limit. The objective of the MCARPTIF is to develop routes of minimum total cost while adhering to all the operational constraints and requirements of the problem.

The MCARPTIF belongs to the class of $\mathscr{NP}$-hard problems, making it impractical to solve large instances, such as those representing actual residential waste collection operations, through exact methods within reasonable computing times. Instead the most relied upon methods to address such problems are through heuristic and metaheuristic solution strategies. In this thesis we develop constructive and Local Search based improvement heuristics, as well as a Tabu Search metaheuristic for the MCARPTIF. A key component of the methods is the trade-off between the quality of the generated solutions and the speed (or time required) to generate and improve the solutions. In this thesis, short, medium and long execution time-limits are imposed, representing practical situations where solutions are sought within three-, thirty and sixty minutes. The performance of the methods is then critically evaluated under these time-limits through benchmark tests on new large MCARPTIF instances.

For the short time-limits four constructive heuristics are developed and tested, and all proved capable of generating feasible solutions for large problem instances within three minutes. A vehicle reduction procedure is also implemented that allows the heuristics to better deal with cases where the fleet size has to be minimised. The performance of the heuristics was inconsistent between different benchmark sets, particularly between waste collection instances and the smaller sets available in literature, thus confirming that the effectiveness of heuristics on small instances does not guarantee that they will perform equally well in more realistic settings. Despite their inconsistent performance, the developed constructive heuristics play an important role in solving the MCARPTIF as they provide initial solutions for more advanced improvement heuristics, which can be applied when more execution time is available.

For medium time-limits we develop advanced Local Search improvement heuristics that rely on two acceleration mechanisms to improve their efficiency. On the largest test instances the basic Local Search setups—that is, without the acceleration mechanisms—took between fifteen minutes and three hours to improve a single solution to local optima. After embedding the accelerated mechanisms within the setups, Local Search took at most four minutes to reach local optima, thus allowing it to be used even when short execution time-limits are imposed.

For long time-limits we extend the accelerated Local Search setup into a deterministic Tabu Search metaheuristic. The metaheuristic takes as input only two parameters, namely a tabu tenure and an execution time-limit. It then improves an initial solution beyond the local optima found using only Local Search. The metaheuristic is tested on large waste collection problem instances and outperforms both the constructive heuristics and accelerated Local Search setups under short, medium and long execution time-limits. The Tabu Search is then further tested as-is on Mixed Capacitated Arc Routing Problem (MCARP) instances and its performance is compared against an existing Memetic Algorithm for the problem. On large instances the Tabu Search is able to outperform the existing method in under three minutes, but on small and medium instances the existing method proved more effective. On these sized instances Tabu Search requires in excess of fifteen minutes and on some instances completely fails to outperform the Memetic Algorithm. Existing heuristic and metaheuristics are thus well capable of dealing with small to medium size instances. However, as our tests on the MCARP instances show, the performance of existing methods on large instances leaves much room for improvement. It is therefore recommended that more tests be performed on large instances such as those introduced in this thesis, which are similar in size to those encountered in practice.

To test the limits of our solution methods a final set of tests are performed on a huge waste collection instance with 6289 required arcs and edges; prior to this thesis the largest instance available from literature only had 803. Two constructive heuristics are capable of generating initial solutions for the instance within three minutes. Thereafter the accelerated Local Search heuristic is able to improve the initial solutions to local optima within thirty minutes. The Tabu Search is then able to marginally improve the solutions within one-hour. More significant improvements are obtained through the Tabu Search when it is allowed up-to 24 hours of execution time, which is expected given the size of the test instance. This final test shows that the heuristics and metaheuristics developed in this thesis are capable of tackling, within reasonable computing times, very large MCARPTIF instances that are similar in size to those encountered in practice.

Keywords: Waste collection; Capacitated Arc Routing Problem, Mixed networks; Intermediate Facilities; Route duration restriction; Splitting procedures; Constructive heuristics; Local search; Metaheuristics.

Acronyms

Problem acronyms

ARP Arc Routing Problem

A routing problem where arcs or edges, typically representing street segments, have to be serviced. Examples of Arc Routing Problems include postal delivery, snow removal, waste collection, and meter-reading.

CARP Capacitated Arc Routing Problem

An Arc Routing Problem where a fleet of vehicles with fixed capacity must service a set of required edges with demand, such that each vehicle route starts and ends at the vehicle depot, and the total demand of edges serviced on a route may not exceed vehicle capacity.

MCARP Mixed Capacitated Arc Routing Problem

The mixed version of the Capacitated Arc Routing Problem where the network to be serviced is mixed. The network can consist of edges that can be serviced or traversed in any direction, as well as arcs that can only be serviced or traversed in one direction.

IF Intermediate Facility

An intermediate facility where a waste collection vehicle is allowed to unload its waste prior to exceeding its capacity and resume servicing required arcs and edges.

CARPIF Capacitated Arc Routing Problem with Intermediate Facilities

An extension of the Capacitated Arc Routing Problem where vehicles are allowed to visit Intermediate Facilities within their routes, as long as the demand serviced in a route between Intermediate Facility visits does not exceed vehicle capacity.

MCARPIF Mixed Capacitated Arc Routing Problem with Intermediate Facilities

The mixed version of the Capacitated Arc Routing Problem with Intermediate Facilities where the network to be serviced is mixed.

CARPTIF Capacitated Arc Routing Problem under Time Restrictions with Intermediate Facilities

An extension of the Capacitated Arc Routing Problem with Intermediate Facilities in which a route-duration limit is imposed on each vehicle route, typically representing the available working hours in a day.

MCARPTIF Mixed Capacitated Arc Routing Problem under Time Restrictions with Intermediate Facilities

The mixed version of the Capacitated Arc Routing Problem under Time Restrictions with Intermediate Facilities.

VRP Vehicle Routing Problem

The node routing equivalent of the Capacitated Arc Routing Problem where a fleet of vehicles with fixed capacity must service a set of service points, instead of edges, such that each vehicle route starts and ends at the vehicle depot, and the total demand of the points serviced on a route may not exceed vehicle capacity.

Algorithm acronyms

M Merge

A constructive heuristic that creates an initial solution by generating a route for each required arc and edge, and then merging the routes until no more mergers are possible without violating constraints. In this thesis we implement a deterministic version of the method, termed *Improved-Merge* (IM), as well as a randomised multi-start version, termed *Randomised-Merge* (RM).

PS Path-Scanning

A constructive heuristic that gradually builds an initial solution by adding the nearest required arc or edge to the current position of a route until the route reaches its capacity or route duration-limit. In this thesis we implement a deterministic version of the method, termed *Path-Scanning* (PS), as well as a randomised multi-start version, termed *Path-Scanning-Random-Link* (PSRL).

RC Route-Cluster

A giant-route based constructive heuristic in which problem constraints are ignored and *Path-Scanning* is used to generate a giant route. The route is then clustered into feasible vehicle routes using optimal splitting procedures. In this thesis we implement a deterministic version of the method, termed *Route-Cluster* (RC), as well as a randomised multi-start version, termed *Route-Cluster-Random-Link* (RCRL).

ERC Efficient-Route-Cluster

An efficient version of *Route-Cluster* which uses a heuristic instead of optimal splitting procedure. In this thesis we implement a deterministic version of the method, termed *Efficient-Route-Cluster* (ERC), as well as a randomised multi-start version, termed *Efficient-Route-Cluster-Random-Link* (ERCRL).

RV Reduce-Vehicles

A vehicle reduction procedure that attempts to reduce the number of routes required for a solution. The procedure removes a route from the solution and attempts to re-insert the arcs of the route into the remaining routes, where feasible, even if it results in an increase in solution cost.

LS Local Search

A greedy heuristic improvement method that improves an initial solution by making small modifications to the solution until a local optimum is reached. In this thesis we implement basic as well as accelerated Local Search versions, referred to as

Local-Search Basic (LS-B), and Local-Search Accelerated (LS-A). LS-B is further tested with a Full (LS-BF) and Reduced Neighbourhood (LS-BR) move neighbourhood, and both versions are tested with a Best-Move (LS-BFB and LS-BRB) and a First-Move (LS-BFF and LS-BRF) move strategy. LS-A is tested with a Reduced (LS-AR) and an Extended (LS-AE) move neighbourhood, and with a Best-Move (LS-ARB and LS-AEB) and Greedy-Independent-Compound-Moves (LS-ARG and LS-AEG) move strategy. All the LS-A setups are further tested with Nearest-Neighbourhood-Lists.

TS Tabu Search

A metaheuristic solution method that uses memory structures to guide Local Search out of local optima. In this thesis we implement an Accelerated Neutral Tabu Search (NATS) that incorporates takes as input two parameters, namely, a tabu tenure and execution time-limit.

Contents

Abstract i

Acronyms iv

Problem acronyms . . . v
Algorithm acronyms . . . vi

Contents ix

List of Figures xiii

List of Tables xvii

List of Algorithms xx

1 Introduction 1

1.1 Municipal solid waste collection and transportation . . . 1
1.2 Capacitated Arc Routing Problems . . . 2
1.3 Problem statement . . . 4
1.4 Research design . . . 5
1.5 Research methodology . . . 6
1.6 Document structure . . . 7

2 Capacitated Arc Routing Problems in literature 9

2.1 Arc routing problems in literature . . . 9
2.1.1 Terms and notations . . . 9
2.1.2 Chinese and rural postmen . . . 9
2.1.3 The Capacitated Arc Routing Problem . . . 10
2.2 Extensions of the Capacitated Arc Routing Problem . . . 12
2.2.1 Mixed networks . . . 12
2.2.2 Intermediate facilities . . . 14
2.2.3 Treatment of vehicle fleet size . . . 14
2.2.4 The MCARPTIF formally defined . . . 16
2.3 Solution approaches for Capacitated Arc Routing Problems . . . 18
2.3.1 Problem difficulty . . . 19
2.3.2 Constructive heuristics . . . 20
2.3.3 Improvement heuristics . . . 20
2.4 Evaluating heuristics . . . 22
2.4.1 Evaluation criteria . . . 22
2.4.2 Problem test instances . . . 23
2.5 Conclusion . . . 28

Chapter appendix 29
2.A Modelling and calculating shortest paths with turn penalties 29
2.B Generic algorithm running time analysis 31
2.C *Cen-IF* problem instance description 31

3 Splitting procedures 33
3.1 Introduction 33
3.2 Splitting procedures for the CARP and CARPTIF 33
3.3 New splitting procedures 35
3.3.1 Splitting procedures for the MCARPIF 35
3.3.2 Splitting procedures for the MCARPTIF 38
3.3.3 Heuristic splitting procedures for the MCARPTIF 42
3.4 Computational results 42
3.5 Conclusion 47

4 Constructive heuristics 49
4.1 Introduction 49
4.2 Constructive heuristics for CARPs 50
4.3 Constructive heuristics for the MCARPTIF 51
4.3.1 Path-Scanning 52
4.3.2 Merge 55
4.3.3 Route-First-Cluster-Second heuristics 59
4.3.4 Vehicle Reduction heuristic 60
4.4 Computational results 61
4.4.1 Evaluation criteria 61
4.4.2 Computational time and break even analysis 64
4.4.3 Performance evaluation 66
4.5 Main findings 73
4.6 Conclusion 73
Chapter appendix 75
4.A Detailed algorithm descriptions 75
4.A.1 Multiple run solution constructor 75
4.A.2 Path-Scanning algorithm 76
4.A.3 Randomised-Merge algorithm 77
4.A.4 Reduce-Vehicles algorithm 79

5 Basic local search heuristics 81
5.1 Introduction 81
5.2 Local search for the CARP and MCARP 82
5.2.1 MCARP move operators 82
5.2.2 Basic acceleration strategies 84
5.3 Basic local search for the MCARPTIF 85
5.3.1 Flip 87
5.3.2 Relocate 89
5.3.3 Exchange 94
5.3.4 Cross 98
5.3.5 Two-opt-1 102
5.4 Computational results 104
5.4.1 Best-move local search 105
5.4.2 First-move local search 108

5.5 Conclusion . . . 110
Chapter appendix . . . 111
5.A First-move versus best-move strategies . . . 111
5.B Additional cross moves for the MCARPTIF . . . 111

6 Accelerated local search heuristics 117
6.1 Introduction . . . 117
6.2 Acceleration mechanisms for the CARP and VRP . . . 117
6.2.1 Nearest neighbour lists . . . 118
6.2.2 Static move descriptors . . . 118
6.2.3 Compounding independent moves . . . 119
6.2.4 Other improvement mechanisms . . . 120
6.3 Accelerated and extended LS for the MCARPTIF . . . 120
6.3.1 Nearest neighbour lists . . . 122
6.3.2 Greedily compounding independent moves . . . 124
6.3.3 Static move descriptors . . . 128
6.3.4 Extending the move neighbourhood . . . 131
6.4 Computational results . . . 134
6.4.1 Analysis of acceleration mechanisms and extended move neighbourhoods . . . 135
6.4.2 Domination analysis . . . 141
6.4.3 Multi-start performance evaluation . . . 145
6.5 Conclusion . . . 154
Chapter appendix . . . 155
6.A Accelerated best-move versus accelerated compounded moves . . . 155
6.B Multi-start analysis . . . 155
6.C Results for multi-start LS setups with nearest-neighbour-lists . . . 157

7 An accelerated tabu search metaheuristic 161
7.1 Introduction . . . 161
7.2 Tabu search for the MCARPTIF . . . 162
7.2.1 Basic instruments . . . 163
7.2.2 Application of efficient local search components within Tabu Search 164
7.2.3 Greedily compounding independent non-tabu moves . . . 165
7.2.4 Accelerated Tabu Search . . . 166
7.3 Computational results . . . 168
7.3.1 Results on the MCARPTIF instances . . . 170
7.3.2 Evaluation against existing solution approaches . . . 174
7.3.3 Computational tests on the *Cen-IF-Full* instance . . . 178
7.4 Conclusion . . . 181
Chapter appendix . . . 183
7.A Tabu tenure parameterisation . . . 183
7.B Results for NATS on the MCARP . . . 183

8 Research contributions and future work 187
8.1 Research aims . . . 187
8.2 Research contributions . . . 189
8.3 Future research opportunities . . . 191

Bibliography 193

Appendices **203**

A Detailed results tables **203**

A.1 Full results for constructive heuristics 203

A.1.1 Results on *Act-IF*, *Cen-IF* and *lpr-IF* 203

A.1.2 Results on small benchmark sets 206

A.2 Full results for the tabu search metaheuristic 220

A.2.1 Results on *Act-IF*, *Cen-IF*, *Lpr-IF* and *mval-IF-3L* 220

A.2.2 Results on *lpr*, *mval* and *bccm-IF* 231

List of Figures

1.1 Network representation of residential waste collection 3

2.1 Example of the Capacitated Arc Routing Problem 11
2.2 Example of the Mixed Capacitated Arc Routing Problem 13
2.3 Example of the Mixed Capacitated Arc Routing Problem with Intermediate Facilities. 15
2.4 Randomly generated *lpr-c-05* like network 24
2.5 *Act-IF* road network 25
2.6 *Cen-IF* road network 25

3.1 Example of a splitting procedure for the MCARPIF 36
3.2 Optimal IF partitions on the example route 39
3.3 Example of a splitting procedure for the MCARPTIF 40
3.4 Results for four MCARPTIF splitting procedures on thirty different giant tours per instance 44
3.5 Results for the cost gap between *Simple-Split* and *Heuristic-Split*, and *Efficient-Split*. 45

4.1 Waste collection area for the recycling case study. 50
4.2 Histogram of cost gaps for *Improved-Merge* and *Randomised-Merge* on the *Cen-IF-b* problem instance. 62
4.3 Problem instance size versus CPU time of deterministic heuristics to produce one solution and for *Reduce-Vehicles* to improve the solution. 64
4.4 Problem instance size versus execution time of *Randomised-Merge*, *Path-Scanning-Random-Link*, *Efficient-Route-Cluster-Random-Link* and *Route-Cluster-Random-Link* implementations. 67
4.5 Results on waste collection benchmark instances of the deterministic and randomised versions of *Merge*, *Path-Scanning* and *Efficient-Route-Cluster* when min Z or min K is the primary objective. 68

5.1 Examples of *relocate*, *exchange*, *two-opt* and *two-opt-1* move operators for the MCARP. 83
5.2 Example of the *flip* move where an arc task in a route is inverted if it has an opposing arc. 87
5.3 Cost of *flip* moves. 88
5.4 Example of the *relocate* move operator. 89
5.5 Cost of the remove component of *relocate* moves. 91
5.6 Cost of the insert component of *relocate* moves. 92
5.7 Conditions for a *relocating* move. 93
5.8 Example of the *exchange* move operator. 95

5.9 Cost of replacing an arc with another arc. 95
5.10 Conditions for *exchanging* arcs $T_{i,j,k}$ and $T_{l,m,n}$. 96
5.11 Example of the *cross* move operator. 99
5.12 The cost and feasibility checks when two routes are *crossed* at positions preceded by arc tasks. 100
5.13 Example of the *two-opt-1* move where the route segment between two arcs is inverted 103
5.14 Results for basic *best-move* LS setups 105
5.15 Cost saving contributions of move operators within the LS-Best and LS-Best-Reduced implementations. 107
5.16 Average time per iteration by each move operator within the LS-Best and LS-Best-Reduced implementations. 107
5.17 Comparison of LS-Basic-Full-Best, LS-Basic-Reduced-Best, LS-Basic-Full-First and LS-Basic-Reduced-First setups on waste collection benchmark sets 109
5.18 Illustration of the difference between *best-move* and *first-move* setups using only the *relocate* move operator on two *Path-Scanning* initial solutions. . . 112
5.19 The cost and feasibility checks when two routes are *crossed* at a position preceded by an arc task, and on the last IF visit in a subtrip. 113
5.20 The cost and feasibility checks when two routes are *crossed* at a position preceded by an arc task, and on the first IF visit in a subtrip. 113
5.21 The cost and feasibility checks when two routes are *crossed* at a position preceded by an arc task, and on the first arc following an IF visit. 114
5.22 Further examples of *cross* moves. 115

6.1 Example of a waste collection area to be serviced with nearest neighbour arc segments. 123
6.2 Four possible improving moves on the example route $\boldsymbol{T}_1$ 126
6.3 Comparison of LS-Accelerated-Reduced-Best, LS-Basic-Reduced-Best and Local Search (LS)-Basic-Full-Best on waste collection benchmark sets . . . 136
6.4 Comparison of LS-Accelerated-Reduced-Greedy, LS-Accelerated-Reduced-Best and LS-Basic-Full-Best on waste collection benchmark sets 137
6.5 Comparison of LS-Accelerated-Reduced-Greedy and LS-Accelerated-Reduced-Best at four *Nearest-Neighbour-Lists* f-levels on waste collection benchmark sets 138
6.6 Move landscape analysis at the first iteration of LS with a reduced neighbourhood on waste collection benchmark sets. 141
6.7 Comparison of LS-Accelerated-Reduced-Greedy, LS-Accelerated-Extended-Greedy, LS-Accelerated-Reduced-Best and LS-Accelerated-Extended-Best on waste collection benchmark sets. 142
6.8 Dominated and non-dominated local search setups in terms of cost savings, fleet reduction and execution time of the setups on waste collection benchmark sets. 143
6.9 Dominated and non-dominated local search setups in terms of cost savings, fleet reduction and execution time of the setups on the *mval-IF-3L* benchmark set. 145
6.10 Cost and fleet size gaps of the multi-start LS-Accelerated-Extended-Best-1 and LS-Accelerated-Extended-Greedy-f setups linked with *Efficient-Route-Cluster-Random-Link*, *Path-Scanning-Random-Link* and *Randomised-Merge* constructive heuristics on *Cen-IF* instances 147

6.11 Cost and fleet size gaps on *Cen-IF* instances of the multi-start and deterministic setups of LS-Accelerated-Reduced-Greedy and LS-Accelerated-Extended-Greedy linked with *Efficient-Route-Cluster*, *Path-Scanning* and *Merge* constructive heuristics, as well as the multi-start randomised constructive heuristic setups on their own. 149

6.12 Cost and fleet size gaps on *Act-IF* and *Cen-IF* instances of the multi-start and deterministic setups of LS-Accelerated-Reduced-Greedy and LS-Accelerated-Extended-Greedy linked with *Efficient-Route-Cluster*, *Path-Scanning* and *Merge* constructive heuristics, as well as the multi-start randomised constructive heuristic setups on their own 151

6.13 Illustration of the difference between LS only using *Static-Move-Descriptors* and using it in combination with *Greedy-Compound-Independent-Moves* . . 156

6.14 Illustration of the multi-start analysis on LS-AEG-0.25 and LS-AEG setups with *Path-Scanning-Random-Link*. 157

6.15 Cost and fleet size gaps of the multi-start LS-Accelerated-Extended-Best-1 and LS-Accelerated-Reduced-Greedy-f setups under execution time-limits on *Cen-IF* instances. 158

6.16 Cost gaps of the multi-start LS-Accelerated-Extended-Best-1 and LS-Accelerated-Extended-Greedy-f setups linked with *Efficient-Route-Cluster-Random-Link*, *Path-Scanning-Random-Link* and *Randomised-Merge* constructive heuristics on *Act-IF* and *Lpr-IF* instances. 159

7.1 Comparison between the best performing Local Search setup per problem instance and *Path-Scanning*, *Improved-Merge* and *Efficient-Route-Cluster* constructive heuristics linked with *Neutral-Accelerated-Tabu-Search* on waste collection benchmark sets 171

7.2 Lower-bound gaps of NATS setups, in comparison to a Memetic Algorithm [6], on nine *Lpr* instances. 177

7.3 Performance of NATS under a 24-hour execution time-limit on the *Cen-IF-Full* instance. 180

7.4 Improvement of *Neutral Accelerated Tabu Search* at different tabu tenure levels to three initial solutions per problem instance. 184

7.5 Cost gap between the NATS setups and the previous best known solutions, in comparison to the Memetic Algorithm of [6], on nine *Lpr* MCARP problem instances. 185

List of Tables

2.1 Glossary of mathematical symbols. 18
2.2 MCARPTIF solution representation symbols. 18
2.3 Benchmark set and problem instance features. 27

3.1 Mean total cost results over 30 experiments for different splitting procedures used in a multi-start *Route-First-Cluster-Second* constructive heuristic . . . 46

4.1 Average heuristic CPU times (in seconds) to generate one solution per benchmark set. 64
4.2 Average number of runs, α_e, required by randomised heuristics to break-even with their deterministic versions. 65
4.3 Number of problem instances on which randomised heuristics failed to break-even over 10000 runs. 66
4.4 Number of problem instances on which *Reduce-Vehicles* was able to reduce the fleet size. 67
4.5 Average Z_{gap} values (in %) on benchmark sets when min Z is the primary or secondary objective. 70
4.6 Average K_{gap} values on benchmark sets when min K is the primary or secondary objective. 71

5.1 Fleet size reduction capabilities of four local search setups on fourteen excessive-fleet initial solutions. 109

6.1 Cost-links of moves between arcs u and v. 127
6.2 Conditions for a move between arcs u and v to break the cost-link (θ_k, θ_{k+1}).127
6.3 The main local search versions and their respective setups tested for the MCARPTIF. 135
6.4 Fleet size reduction, ΔK_{LS}, of LS-ARG-f and LS-ARB-f setups on fourteen excessive-fleet initial solutions. 140
6.5 Averages for the expected percentage cost gaps, $\overline{Z}_{\text{gap}}$ %, of heuristic setups under short-term and medium-term and planning execution time-limits of three and thirty minutes, respectively. 152
6.6 Number of instances on which the of heuristic setups failed to produce solutions with the known minimum fleet size. 153

7.1 Average cost gaps, Z_{gap} in %, of different optimisation algorithms under short, medium and long execution time-limits on MCARPTIF benchmark sets. 173

7.2 Number of problem instances per MCARPTIF benchmark set on which the different optimisation algorithms under short, medium and long execution time-limits produced excess-fleet solutions. 175
7.3 Average lower-bound and best-known cost gaps for NATS setups on MCARP and CARPTIF benchmark sets. 177
7.4 Performance of the optimisation algorithms on the *Cen-IF-Full* instance. . . 179

A.1 Heuristic results for *Act-IF*, *Cen-IF* and *Lpr-IF* sets 204
A.2 Computational times, in seconds, of heuristics to complete 100 runs (α = 100) for *Cen-IF* and *Lpr-IF*, *Act-IF* sets. 206
A.3 Heuristic results for *mval-IF-3L* set 207
A.4 Heuristic results for *bccm-IF-3L* set 209
A.5 Heuristic results for *gdb-IF-3L* set 211
A.6 Heuristic results for *bccm-IF* set 212
A.7 Heuristic results for *gdb-IF* set 215
A.8 Computational times, in seconds, of heuristics for *mval-IF-3L*. 217
A.9 Computational times, in seconds, of heuristics for *bccm-IF-3L*. 217
A.10 Computational times, in seconds, of heuristics for *gdb-IF-3L*. 218
A.11 Computational times, in seconds, of heuristics for *bccm-IF*. 218
A.12 Computational times, in seconds, of heuristics for *gdb-IF*. 219
A.13 Expected cost, Z, for the constructive heuristic, accelerated Local Search, and Neutral Accelerated Tabu Search setups under different time-limits on the *Cen-IF*, *Lpr-IF*, *Act-IF*, and *mval-IF-3L* sets. 221
A.14 Expected fleet size, K, for the constructive heuristic, accelerated Local Search, and Neutral Accelerated Tabu Search setups under different time-limits on the *Cen-IF*, *Lpr-IF*, *Act-IF*, and *mval-IF-3L* sets. 225
A.15 Cost results for the Memetic Algorithm (MA) of [6] and for the multi-start and deterministic Neutral Accelerated Tabu Search (NATS) setups linked with different constructive heuristics on the *lpr* set when allowed 60 minutes of execution time. 231
A.16 Cost results for the Memetic Algorithm (MA) of [6] and for the multi-start and deterministic Neutral Accelerated Tabu Search setups linked with different constructive heuristics on the *mval* set when allowed 60 minutes of execution time. 232
A.17 Cost results for the Variable Neighbourhood Search (VNS) algorithm of [68] and for the multi-start and deterministic Neutral Accelerated Tabu Search setups linked with different constructive heuristics on the *bccm-IF* set when allowed 60 minutes of execution time. 233

List of Algorithms

2.1 *Arc-to-Arc-Floyd-Warshall* algorithm with turn penalties . . . 30
2.2 *Shortest-Path-Arc-Sequence* . . . 30

3.1 *Efficient-IF-Split* for the MCARPIF . . . 37
3.2 *Efficient-Split* for the MCARPTIF . . . 41
3.3 *Heuristic-Split* for the MCARPTIF . . . 43

4.1 *nConstruct* . . . 75
4.2 *Path-Scanning* . . . 76
4.3 *Initialise-Merge* . . . 77
4.4 *Execute-Merge* . . . 78
4.5 *Insert-Arc* . . . 79
4.6 *Reduce-Vehicles* . . . 80

5.1 *Generic-Local-Search* . . . 82
5.2 *Basic-Local-Search* . . . 86
5.3 *Find-Flip-Subtrip-Move* . . . 88
5.4 *Find-Flip-Move* . . . 89
5.5 *Find-Relocate-Subtrip-Move* . . . 92
5.6 *Find-Relocate-Move* . . . 94
5.7 *Find-Exchange-Subtrip-Move* . . . 97
5.8 *Find-Exchange-Move* . . . 98
5.9 *Find-Cross-Subtrip-Move* . . . 101
5.10 *Find-Cross-Move* . . . 101
5.11 *Find-Two-Opt-1-Move* . . . 104

6.1 *Move-Cost* . . . 121
6.2 *Check-Feasibility* . . . 122
6.3 *Find-Relocate-Moves* . . . 123
6.4 *Find-Exchange-Moves* . . . 124
6.5 *Find-Cross-Moves* . . . 125
6.6 *Greedy-Compound-Independent-Moves* . . . 128
6.7 *Update-Move-Dependence-Arc-Sets* . . . 130
6.8 *Check-Move-Independence* . . . 130
6.9 *Greedy-Compound-Moves* . . . 131
6.10 *Update-Savings-List* . . . 132
6.11 *Accelerated-Local-Search* . . . 132

7.1 *Tabu-Search* . . . 163
7.2 *Tabu-Greedy-Compound-Moves* . . . 165

7.3 *Update-Tabu-List* . . . 166
7.4 *Accelerated-Tabu-Search* . . . 167
7.5 *Neutral-Tabu-Greedy-Compound-Moves* . . . 168
7.6 *Neutral-Accelerated-Tabu-Search* . . . 169

Chapter 1

Introduction

Waste management is an important basic service provided by local municipalities, and consists of the effective management of waste from its creation to its final disposal. A key activity of waste management is the collection and transportation of waste to landfill sites. This seemingly simple task is one of the most costly of the waste management functions, and the scale at which it is performed makes it a promising area to target for improvement. This thesis deals with the optimisation of waste collection vehicle routes to improve the collection and transportation function of local municipalities. By treating waste collection routing as an arc routing problem, heuristics are developed that can be used to generate and improve collection routes for large collection areas, consistent in size with those serviced by local municipalities.

1.1 Municipal solid waste collection and transportation

Solid waste collection and transportation consists of the collection, transportation and disposal of waste at landfill sites, usually through waste collection vehicles. It is well recognised as being the most costly component of the waste management function and can account for between 50-80% of a municipality's solid waste management budget [49, 84]. In 2010, South African municipalities spent R7.3 billion[1] on solid waste management, and it is estimated that waste collection transportation costs amounted to 45% of this total [65]. Waste collection and transportation is assigned to and managed by 278 local government entities. At this sub-level the scale of waste collection operations can still be massive. As a case in point, the City of Cape Town, which is the second largest metro in South Africa, manages a collection fleet of 887 vehicles, of which more than 200 are used to service the approximately 1 000 000 households of the City on a weekly basis [16].

The state of practice method of residential waste collection is through curb side collection. Municipalities have to collect the waste of each household at least once a week. Households place their generated waste, which is stored in either bins or bags, on the designated days in front of their properties where waste collection vehicles can then collect the waste. This process is highly repetitive and performed throughout the year. In 2010, truck costs were responsible for R1.9 billion and fuel costs for R1.4 billion of the total waste collection expenditures in South Africa, therefore even a small improvement in waste collection and transfer operations can lead to significant savings in costs. A promising improvement area is to design better waste collection routes, with the aim to minimise the number of vehicles to service a specific area, and to minimise the total distance travelled

[1] This is equal to US$ 947 million, calculated using the average US$ 1 = R 7.5 exchange rate of 2010.

by the vehicles. This led us to the following broad research question:

> *How should municipalities design and optimise residential waste collection routes for their waste collection vehicles?*

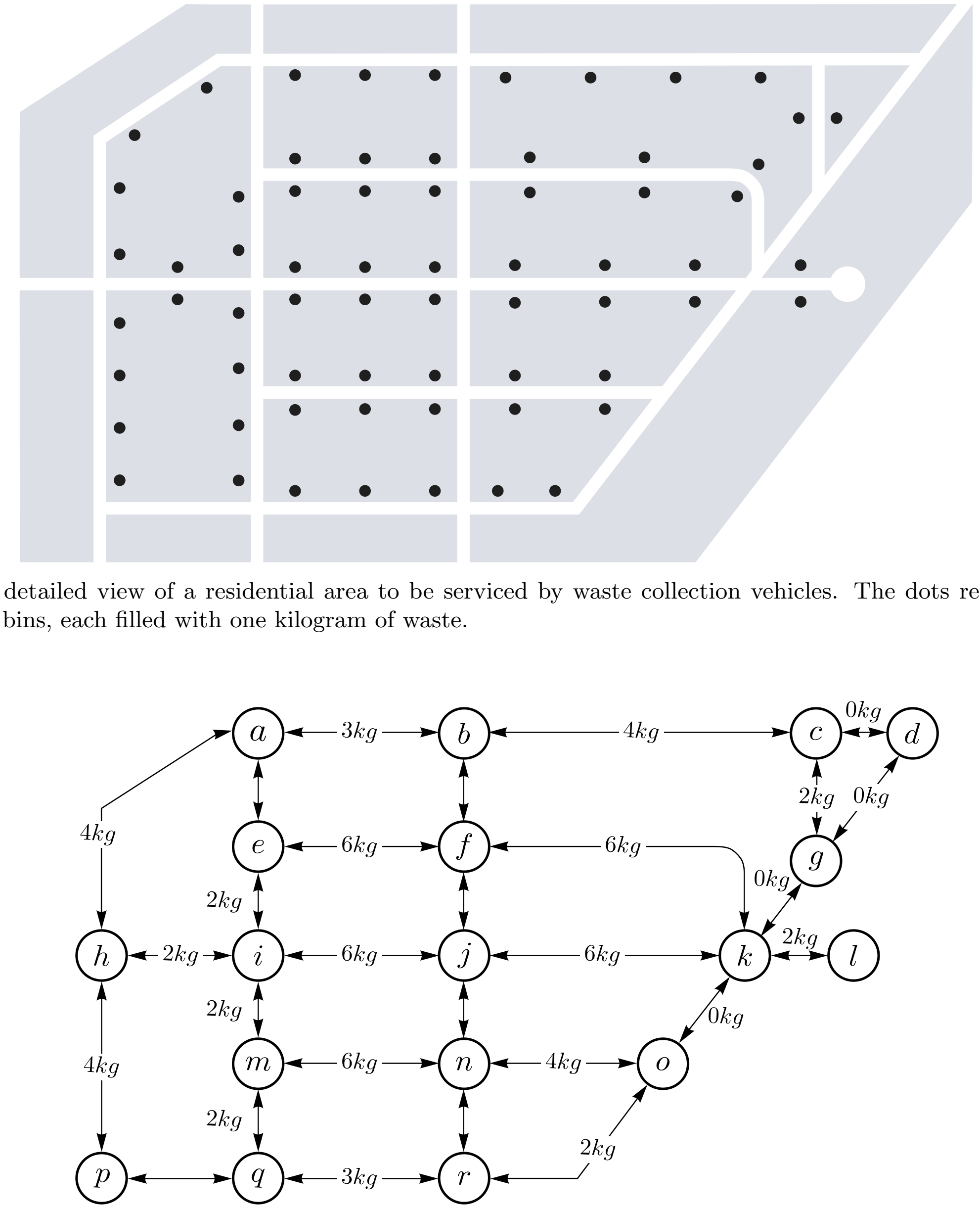
To better answer the research question the next section illustrates how residential waste collection can be simplified using network modelling. Through the simplification we show that designing collection routes can be modelled as a network optimisation problem known as the Capacitated Arc Routing Problem (CARP).

1.2 Capacitated Arc Routing Problems

Illustrated in Figure 1.1a is an example of an urban area where the dots represent residential waste bins. Since all of the area's waste has to be collected, each of the 65 bins of Figure 1.1a has to be visited by collection vehicles. The problem can be simplified as follows. Instead of stating that each bin has to visited by a collection vehicle, we rather state that each road segment has to be serviced. Given this aggregation the road network in Figure 1.1a can be simplified to the network in Figure 1.1b where each of the seventeen vertices, a through r, represent a street intersection or dead-end of Figure 1.1a. Each road segment connecting two vertices is classified as an edge, and since all roads are two way streets the edges can be serviced in either direction. Finally, the waste to be collected on each edge is referred to as the demand of the edge.

Network problems that require road segments (arcs and edges) to be serviced are known in literature as Arc Routing Problems (ARPs). When a capacity constraint is enforced, which in our case is the amount of waste a vehicle can collect before it is full, the problem generalises to the Capacitated Arc Routing Problem (CARP), first proposed by Golden and Wong [39]. The CARP has a wide range of applications such as security guard routing [85] and railway maintenance [43], and its well known applications of snow plowing [13] and waste collection [35], among others. The objective of the problem is to determine routes of minimal total cost for a fleet of homogeneous vehicles so that each route starts and ends at the vehicle depot, each road segment with demand is serviced exactly once by a vehicle, and the sum of demand serviced by a route does not exceed vehicle capacity. The fleet size can be either limited, unlimited, minimised or left as a decision variable. Total cost can also be measured in total distance travelled or the sum of the time taken to complete all routes.

In terms of waste collection, various successful CARP applications can be found in literature. Bautista et al. [4] use a CARP transformation to design collection routes for Sant Boi de Llobregat within the metropolitan area of Barcelona (Spain). The new routes result in a 22 000 km yearly route length reduction. Consequently fuel consumption is reduced and additional time of truck drivers is made available to perform other tasks such as vehicle maintenance and street sweeping. Ghiani et al. [30] conduct a solid waste collection study for the Municipality of Castrovillari, a town located in Southern Italy. The authors model waste collection as a variant of the CARP and their routing solutions result in overall annual savings of more than 13 000 Euros per year. The authors estimate that applying their analysis in other Italian municipalities could save hundreds of millions of Euro annually. Sahoo et al. [74] report on the development of WasteRoute, a comprehensive route-management system for waste collection. The system was developed for Waste Management, Inc. who provide residential waste collection services. The system employs CARP modelling and within one year of its inception the company had 984 fewer routes, resulting in annual savings of eighteen million US dollars. Based on the successful CARP



(a) A detailed view of a residential area to be serviced by waste collection vehicles. The dots represent waste bins, each filled with one kilogram of waste.

(b) A simplified road network representation of the residential area. Each street intersection and dead-end is modelled as a vertex, and each street segment between two vertices is modelled as an edge. The demand, in kilogram, of an edge is the cumulative amount of waste on the original street segment.

Figure 1.1: Network representation of an urban area that has to be serviced by waste collection vehicles. The top figure shows a birds-eye view of a service area. The bottom figure shows the simplified network representation of the same area.

implementations, enough motivation exists to model and solve the problem of designing residential collection routes for municipalities as a CARP.

1.3 Problem statement

The research question that this thesis answers is how should residential waste collection be modelled and solved to enable municipalities to design and optimise collection routes for their waste collection vehicles? Collection routes are part of the operational component of Solid Waste Management. Designing the routes is defined by Ghiani et al. [35] as the solid waste collection problem and it consists of the following key operational components:

- every route starts and ends at a depot, and at the end of its route the vehicle must arrive empty at the depot in the event that it does not coincide with a landfill site;
- on every route the total amount of waste collected between the depot and the first visit to a landfill site, or between two successive visits to landfill sites must not exceed vehicle capacity; and
- the duration of any route cannot exceed a maximum shift length, in accordance with local rules and regulations.

An implicit component that we formally add is that:

- collection vehicles must adhere to road restrictions.

This is enforced to ensure that the resulting routes take into account one-way streets that can only be traversed in a specific direction, and that the routes do not require illegal turns.

Two CARP extensions deal with the four key components of the solid waste collection problem. The Mixed Capacitated Arc Routing Problem (MCARP), first proposed by Lacomme et al. [51], models road networks with two-way streets that require zigzag collection, two-way streets with sides requiring separate collecting, and one-way streets that can only be traversed and serviced in one direction. The Capacitated Arc Routing Problem under Time Restrictions with Intermediate Facilities (CARPTIF), introduced by Ghiani et al. [31][2], models the case where vehicles may unload their collected waste at one of multiple Intermediate Facilities (IFs) such as dumpsites, return to a service area and continue collecting waste. The sum of demand collected on a subtrip between Intermediate Facility (IF) visits may then not exceed vehicle capacity and the route must include a final IF visit before returning to the depot. Furthermore, the duration of a vehicle route may not exceed a time restriction, typically equal to the available working hours per shift.

In this thesis we extend the CARPTIF to the Mixed Capacitated Arc Routing Problem under Time Restrictions with Intermediate Facilities (MCARPTIF), thus accounting for the mixed road networks of the MCARP. As is often the case for municipalities, we focus on large MCARPTIFs that require routes to be developed to service thousands of street segments. To solve large MCARPTIFs, and in so doing answer the research question, is the main objective of this thesis.

The novelty of our research lies in two areas. According to our knowledge, this thesis represents the first formal study on solution methods for the MCARPTIF, which accurately models residential waste collection. We thereby close the research gap between

[2] Ghiani et al. [31] call the problem the Arc Routing Problem under Capacity and Lengths Restrictions with Intermediate Facilities, which they abbreviate CLARPIF. For waste collection applications, the length restriction is replaced by a time-based route duration restriction, thus why we termed it the CARPTIF.

the waste collection problems faced by municipalities and the ARPs studied in literature. Our focus on large instances necessitated us to focus on and develop efficient solution approaches. In doing so we addressed the CARP research priority recently proposed by Prins [69], which is to develop advanced solution approaches that are able to efficiently tackle huge instances met in real applications.

1.4 Research design

This dissertation primarily employs concepts and tools from the discipline of Operations Research to solve the MCARPTIF. Operations Research deals with decision problems by formulating and analysing mathematical models [71], and central to the discipline is optimisation, both as a solution tool and a modelling device [66]. The MCARPTIF is an optimisation problem with the objective to find the best solution from all feasible solutions.

Since the CARP and all its extensions are $\mathscr{NP}$-hard the most effective way of dealing with the problems are through heuristics. Constructive heuristics are first used to find feasible solutions for the problem. The solutions are then improved during subsequent phases using advanced improvement heuristics. The type and structure of the heuristics are highly dependent on the CARP variant and its formulation. Thus, the first aim of this thesis is to:

(1) Find the appropriate Arc Routing Problem formulation for the MCARPTIF.

Inherent to heuristics is the trade-off between the quality of the generated solutions and the speed or time required to generate and improve the solutions. The longer a heuristic is allowed to execute, the more likely that the heuristic will find a good solution. Conversely, if a heuristic is only afforded a limited amount of execution time, chances are the final solution will be of inferior quality. Finding the correct trade off between speed and quality is by no means simple and depends on the problem environment. To illustrate the concept consider the following three examples.

Example 1: A residential waste collection manager wishes to determine the collection routes for the next six months. Each collection route will then be assigned to a collection crew.

In this example high quality solutions are desirable and the algorithms may be executed for an extended period of time to generate high quality collection routes. Designing collection routes for the next six months is a long term decision. Hopefully the manager has allowed sufficient planning time before the routes have to implemented. The algorithms will probably be executed more than once during the planning phase since the final collection routes have to be critically evaluated by the manager before being implemented. For this reason an execution time-limit may still be imposed.

Example 2: Upon arrival at work, a municipal manager is informed that two of the five waste collection vehicles are inoperable and cannot be used for the day's waste collection. The three remaining vehicles need to be assigned new collection routes that service the whole area. The manager needs new collection routes, and needs them as soon as possible. The longer the three crews wait for new routes, the longer the crews will have to work over-time.

In this example the algorithms need to generate new routes as quickly as possible. The routes will only be used for a day, making inaction more expensive than implementing an inferior solution.

Example 3: The municipal manager is informed that the two collection vehicles will only be operable within three days. The three remaining vehicles need to be assigned new routes for the next three days.

For this example more time is available to generate new collection routes. The algorithms cannot execute for hours on-end, hence the routes will be inferior to the long term decision routes of Example 1, but they will still be better than the short term routes of Example 2.

Each example involves the exact same problem: the MCARPTIF. Yet, with each the emphasis is either more towards solution quality or more towards solution speed. For heuristics to be useful in practise they should be able to act as either tortoise or hare, depending on what the situation requires. There is, however, a risk that the heuristics are afforded unlimited execution time and still produce mediocre solutions. Thus, the developed heuristics have to be critically evaluated in terms of solution speed and solution quality. Accordingly the second, third and fourth aim of the thesis are to:

(2) Identify potential solution methods for the MCARPTIF under different execution-time-limits.

(3) Develop heuristics capable of generating and improving feasible solutions for the MCARPTIF under different execution time-limits.

(4) Critically evaluate the heuristics in terms of their solution quality and execution times.

For the thesis we define three execution time-limit categories, namely short, medium and long execution-times, and limited each to three, thirty and sixty minutes, respectively. Time-limits are not always strictly enforced since the goal of the thesis is to investigate the speed-quality trade-off, and to use the analysis to identify the best heuristics for short, medium and long execution times.

In summary, the main deliverables of this thesis are a formal ARP based formulation for the MCARPTIF, as well as heuristics capable of generating routes within short, medium and long execution times. The successful CARP applications in [4, 30, 74] all employ models and algorithms designed using Operations Research tools and concepts. As such, the research methodology of this thesis is based on the Design Research paradigm, as defined for Operations Research by Manson [56].

1.5 Research methodology

Manson [56] defines Design Research as:

> *"...a process of using knowledge to design and create useful artefacts, and then using various rigorous methods to analyse why, or why not, a particular artefact is effective. The understanding gained during the analysis phase feeds back into and builds the body of knowledge of the discipline."*

In this thesis the created artefacts are the heuristics, in the form of computer algorithms that are capable of computing feasible solutions for the MCARPTIF. Designing the algorithms, of itself, cannot be considered research. However, the process of using knowledge to design the algorithms, and then systematically and rigorously analysing the effectiveness of the algorithms is considered research.

Manson [56] further gives a six phase Design Research methodology. The process starts with *problem awareness* whereby a researcher identifies a potential problem worth investigating. The MCARPTIF, with its application in waste collection and the potential benefits of improving this function for municipalities, qualifies as such a problem. In the second phase, called *suggestion*, one or more tentative designs are developed. We first formulate the MCARPTIF, and use the formulation to identify possibles solution approaches for the problem. Thereafter we identify heuristics, from literature, that have been successfully applied to the CARP and its extensions, and that can be adapted for the MCARPTIF.

The third and fourth phases of the methodology are the main focus of this thesis. During the *development* phase, the identified heuristics are adapted to the MCARPTIF and implemented as computer algorithms. For short execution times, algorithms are developed that quickly construct initial feasible solutions. For medium execution times, algorithms are developed that take the initial solutions as input and improve them using efficient improvement heuristics. For long execution times, the improvement heuristics are used as building blocks to develop a metaheuristic algorithm that further improves the initial solutions. For the *evaluation* phase we use benchmark problem instances to rigorously evaluate the algorithms. The choice of test instances is critical as their structure can influence algorithm performance. In this thesis we develop benchmarks for the MCARPTIF, which include large waste collection instances based on actual road networks. The algorithms are critically evaluated on these benchmarks in terms of solution speed and solution quality.

In the last *conclusion* phase we establish application boundaries for our algorithms. A very large MCARPTIF instance, whose size is consistent with the huge instances met in real applications, is solved and the algorithms are critically evaluated.

Manson [56] gives three minimum requirements for a research project to be considered design research. First, the project must produce one or more artefacts. Second, to determine if the research contributions are at all significant, the following two questions, originally posed by Hevner et al. [46], have to be answered:

> *"What utility does the new artefact(s) provide?"* and *"What demonstrates this utility?"*

In the last part of this thesis we critically evaluate our research contributions and answer the above two questions.

1.6 Document structure

In the next chapter we review variants of the CARP that are applicable to waste collection routing, and review potential solution methods for the problems. More in-depth and technical reviews, focussing on specific solution methods, are included in the subsequent chapters. In Chapters 3 and 4 we develop constructive heuristics that can quickly generate feasible solutions. The focus of Chapters 5 and 6 is on local search heuristics capable of improving the constructive heuristic solutions when more execution time is available. Lastly in Chapter 7, we develop a local search based metaheuristic that is capable of further improving solutions. Each chapter starts with a technical review of applicable solution methods, followed by a description of our developed heuristics. Thereafter, in each chapter, we report on the computational tests performed with the heuristics. The thesis is concluded in Chapter 8 with a summary of our main research contributions and suggestions for future work.

Chapter 2

Capacitated Arc Routing Problems in literature

The aim of this literature review is threefold: firstly, to identify the most appropriate Arc Routing Problem (ARP) formulation for the Mixed Capacitated Arc Routing Problem under Time Restrictions with Intermediate Facilities (MCARPTIF); secondly, to use the ARP formulation and identify appropriate heuristics, applicable to the MCARPTIF; and lastly, to identify validation methods for the heuristics. This chapter only focus on literature relevant to residential waste collection routing. For a comprehensive review of ARPs the reader is referred to [18, 19, 25].

2.1 Arc routing problems in literature

Residential waste collection requires waste to be collected on a street-by-street basis. As such, the problem of designing collection routes can be modelled as an ARP. This section reviews three basic ARPs. The first problem, the Chinese Postman Problem, is expanded to the Rural Postman Problem, which, in turn, is expanded to the Capacitated Arc Routing Problem (CARP). The CARP forms the basis of our MCARPTIF formulation. Before discussing the various problems, we first introduce standard ARP terminology and notations used in this chapter.

2.1.1 Terms and notations

From [26], we let $\boldsymbol{G} = (\boldsymbol{V}, \boldsymbol{A})$ be a connected graph without loops where $\boldsymbol{V} = \{v_1, \ldots, v_n\}$ is the vertex set (or node set), and $\boldsymbol{A} = \{(v_i, v_j) : v_i, v_j \in \boldsymbol{V} \text{ and } i \neq j\}$ is the arc set. With every arc (v_i, v_j) is associated a non-negative cost, distance or length d_{ij}; assuming that $d_{ij} = \infty$ if (v_i, v_j) is not defined. The matrix $\boldsymbol{D} = (d_{ij})$ is symmetric if and only if $d_{ij} = d_{ji}$ for all i and j. When $\boldsymbol{D}$ is symmetric, it is common to associate an *edge* with every vertex pair. Depending if $\boldsymbol{D}$ is symmetric or asymmetric the associated ARP is termed either *undirected* or *directed*. For *undirected* ARPs, the graph $\boldsymbol{G} = (\boldsymbol{V}, \boldsymbol{A})$ can be replaced with $\boldsymbol{G} = (\boldsymbol{V}, \boldsymbol{E})$ where $\boldsymbol{E} = \{(v_i, v_j) : v_i, v_j \in \boldsymbol{V} \text{ and } i < j\}$.

2.1.2 Chinese and rural postmen

Two basic and important ARPs can be derived from general routing problems. These are the Chinese Postman Problem and the Rural Postman Problem. An overview of the two problems is given in [19, 34, 52]. For the Chinese Postman Problem the complete edge set $\boldsymbol{E}$ has to be traversed by a single postman, and the objective is to find a closed tour of minimum length that traverses all the edges $\boldsymbol{E}$ of the graph $\boldsymbol{G} = (\boldsymbol{V}, \boldsymbol{E})$. For the Rural Postman Problem there are a number of villages whose required set $\boldsymbol{R}$ of streets (with $\boldsymbol{R} \subseteq \boldsymbol{E}$) has to be serviced by a postman, and a set $\boldsymbol{E} \backslash \boldsymbol{R}$ of links between villages that do not have to be serviced, but may be used for travelling between villages [26]. Note that the Rural Postman Problem transforms into the Chinese Postman Problem if $\boldsymbol{R} = \boldsymbol{E}$. The objective of the Rural Postman Problem is to find a closed tour of minimum length that traverses all the edges $\boldsymbol{R}$, referred to as *required* edges, of the graph $\boldsymbol{G} = (\boldsymbol{V}, \boldsymbol{E}, \boldsymbol{R})$.

Of the two problems, the Rural Postman Problem is most similar to residential waste collection, primarily because it does not require the whole set $\boldsymbol{E}$ of edges to be serviced. However, both the Rural Postman Problem and Chinese Postman Problem only consider a single postman, as such there are is limit on the length of the single Rural Postman Problem and Chinese Postman Problem tours. This is seldom the case in reality. By introducing capacity constraints and multiple postmen, or vehicles, more realistic ARPs known as CARPs can be formulated.

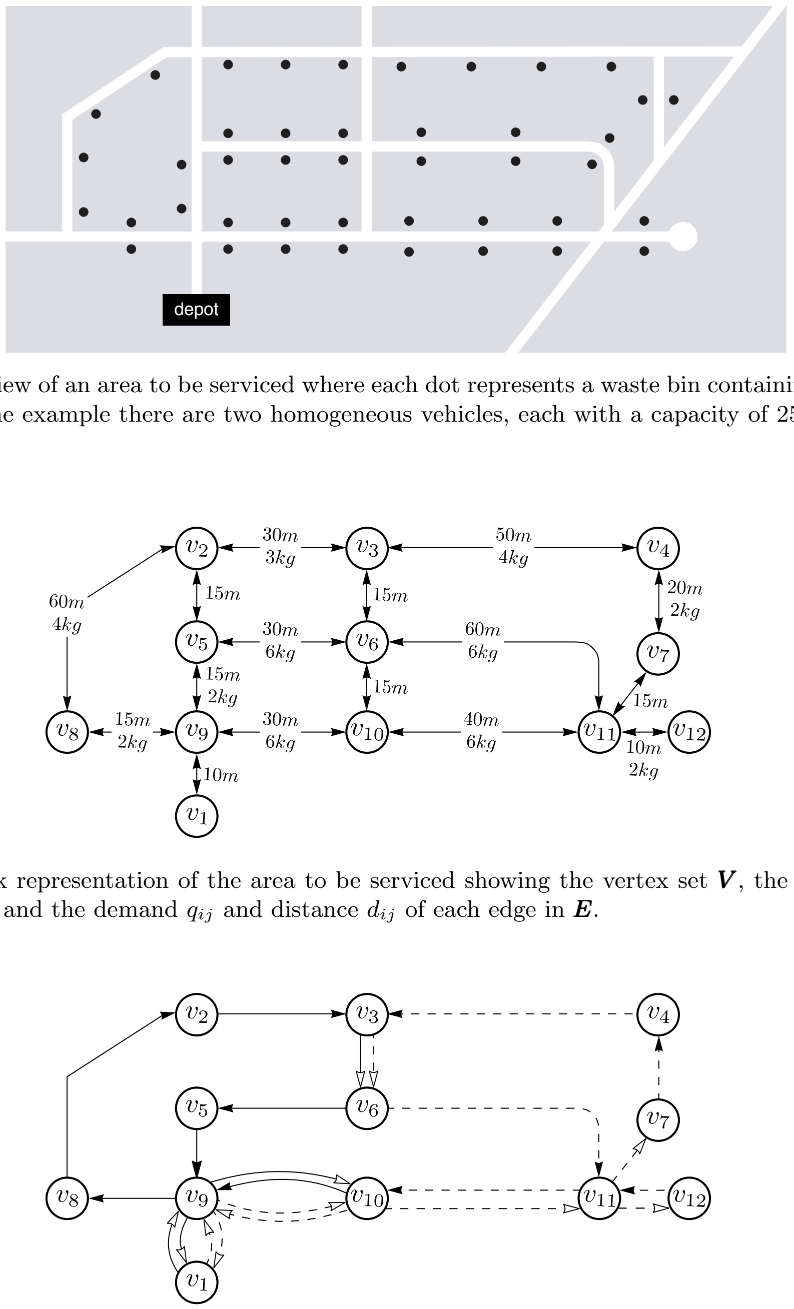
2.1.3 The Capacitated Arc Routing Problem

As most practical routing applications contain capacity restrictions, the CARP, introduced by Golden and Wong [39], is probably the most important problem in the area of arc routing [27]. Examples of the CARP include the routing of street sweepers, snow removal vehicles, and of course waste collection vehicles. For comprehensive reviews dedicated to the problem we refer the reader to [69, 91].

For the CARP each edge (v_i, v_j) has a nonnegative demand or weight q_{ij}. It is assumed that a fleet of K homogeneous vehicles, each with capacity Q, are based at a depot located at vertex v_1. The fleet size K can be either limited, unlimited, minimised or left as a decision variable. The set $\boldsymbol{R}$ of edges (v_i, v_j) with $q_{ij} > 0$, referred to as *required* edges, must be serviced, but the remaining *non-required* edges may also be traversed, which is commonly called *dead-heading*. The CARP consists of designing vehicle routes of total minimal length so that each route starts and ends at the depot, each edge with a demand is serviced exactly once by a single vehicle, and the sum of demand on any route does not exceed Q.

Figure 2.1a shows an example of the CARP on a simple road network with two-way streets and a single vehicle depot. Dots represent bins containing one kilogram of waste. The problem can be modelled as a graph $\boldsymbol{G} = (\boldsymbol{V}, \boldsymbol{E}, \boldsymbol{R})$ as shown in Figure 2.1b. The figure also gives the demand, q_{ij}, and distance or cost, d_{ij}, for all the edges. For the example there are two homogeneous vehicles with a capacity of 25 kilograms each, situated at the depot vertex v_1. Figure 2.1c shows a feasible solution for the CARP. With the solid-line route a vehicle will collect 23 kg of waste and travel 185 meters. A vehicle assigned to the dashed-line route will collect 20 kg of waste and travel 330 meters. The total cost of the solution, i.e., its objective value, is 515 meters.

Real residential waste collection problems cannot be approached exactly as a CARP due to specific operational conditions and constraints [76]. The CARP considers only undirected networks, whereas road networks may 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 streets that can be serviced in either direction and both sides simultaneously. The CARP also does not consider vehicle offloads at dumpsites. In practice, vehicles may unload their collected waste at one of multiple Intermediate



(a) A detailed view of an area to be serviced where each dot represents a waste bin containing one kilogram of waste. For the example there are two homogeneous vehicles, each with a capacity of 25 kg, situated at the depot.

(b) The network representation of the area to be serviced showing the vertex set $\boldsymbol{V}$, the edge set $\boldsymbol{E}$, the depot vertex v_1 and the demand q_{ij} and distance d_{ij} of each edge in $\boldsymbol{E}$.

(c) Two vehicle routes servicing the area. A solid arrow-head indicates that a vehicle services that edge, whereas an empty arrow-head indicates that a vehicle only traverses that edge, also known as dead-heading. With the solid-line route a vehicle will collect 23 kg of waste and travel a distance of 185 meters, and with the dashed-line route a vehicle will collect 20 kg of waste and travel 330 meters.

Figure 2.1: An example of the Capacitated Arc Routing Problem with the objective to design vehicle routes of total minimal length.

Facilities (IFs)[1], return to the service area and continue collecting waste. The sum of demand collected on a subtrip between Intermediate Facility (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 duration of a vehicle route may not exceed a time restriction, typically equal to the available working hours per shift. Two realistic extensions of the CARP that take mixed networks and IFs into account are discussed next.

2.2 Extensions of the Capacitated Arc Routing Problem

This section reviews the most relevant extensions of the CARP that address the shortcomings mentioned previously. Two extensions are considered, namely the CARP on a mixed network, which models one-way and two-way streets and the CARP with IFs that accounts for multiple dumpsites. The two extensions form the basis of the MCARPTIF.

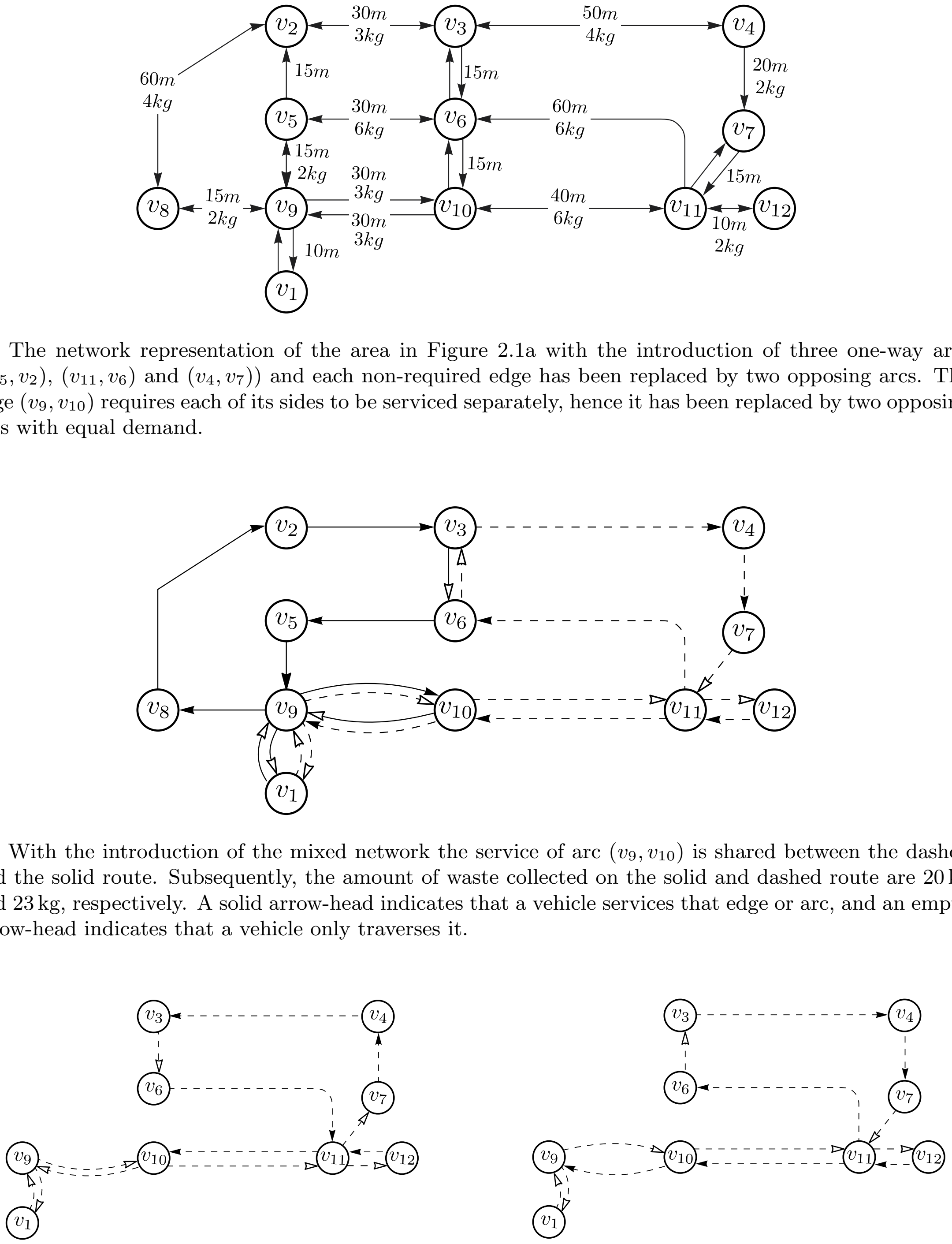
2.2.1 Mixed networks

The Mixed Capacitated Arc Routing Problem (MCARP), studied in [6, 21, 40, 51], among others, allows the modelling of more realistic street networks. The MCARP models one way streets as directed arcs and two way streets as undirected edges. The graph $\boldsymbol{G}$ is thus extended to $\boldsymbol{G} = (\boldsymbol{V}, \boldsymbol{E} \cup \boldsymbol{A})$ where the set of arcs $\boldsymbol{A} = \{(v_i, v_j) : v_i, v_j \in \boldsymbol{V} \text{ and } i \neq j\}$ models one way streets. Real waste collection routing problems usually involve mixed road networks [4, 23, 30, 35, 75] making the MCARP an important practical variant of the CARP.

The extension allows for the modelling of two kinds of non-required streets and four kinds of required streets. Any non-required street is modelled either as one arc, if the street is one-way, or two opposing arcs if the street is two-way. A required street can be a two-way street with both sides serviced simultaneously, modelled as one edge; a two-way street with sides requiring separate collecting, modelled as two opposing arcs; or a one-way street, modelled as one arc. The graph can be further expanded to cater for even more complicated cases. For instance, two parallel arcs can model a one-way street too wide for simultaneous collection and requiring two traversals, one for each side.

Figure 2.2 illustrates the impact of a mixed network by extending the road network of the example in the previous section. Figure 2.2a sees the introduction of two required arcs, (v_4, v_7) and (v_{11}, v_6), and a non-required arc, (v_5, v_2). The example also has a required edge, (v_9, v_{10}), that is too wide for simultaneous collection and requires each side to be collected separately. Accordingly, this edge is modelled as the two opposing arcs, (v_9, v_{10}) and (v_{10}, v_9), and the demand of the edge is divided between the two arcs. Lastly, all the non-required edges are modelled as two opposing arcs. A feasible solution for the MCARP example is shown in Figure 2.2b. With the solution the service of edge (v_9, v_{10}) is shared between the two routes and the waste collected on the solid and dashed routes are 23 kg and 20 kg, respectively. The mixed network also requires a change in the vertex visitation sequence of the dashed route (Figure 2.2c).

[1] In general, intermediate facilities refer to waste transfer stations where collection vehicles unload their waste. Dedicated bulk carriers then transport the waste to dumpsites. For route design purposes we collectively refer to any site where a vehicle can unload its waste as an intermediate facility. Hence, intermediate facilities may include dumpsites and the vehicle depot.



(a) The network representation of the area in Figure 2.1a with the introduction of three one-way arcs $((v_5, v_2)$, (v_{11}, v_6) and $(v_4, v_7))$ and each non-required edge has been replaced by two opposing arcs. The edge (v_9, v_{10}) requires each of its sides to be serviced separately, hence it has been replaced by two opposing arcs with equal demand.

(b) With the introduction of the mixed network the service of arc (v_9, v_{10}) is shared between the dashed and the solid route. Subsequently, the amount of waste collected on the solid and dashed route are 20 kg and 23 kg, respectively. A solid arrow-head indicates that a vehicle services that edge or arc, and an empty arrow-head indicates that a vehicle only traverses it.

(c) Because of the (v_{11}, v_6) and (v_4, v_7) one-way streets the dashed route's vertex visitation sequence is changed from [1, 9, 10, 11, 12, 11, **7**, **4**, **3**, **6**, 11, 10, 9, 1], as shown in the left figure, to [1, 9, 10, 11, 12, 11, **6**, **3**, **4**, **7**, 11, 10, 9, 1], as shown in the right figure. The length of both routes are the same.

Figure 2.2: An example of the Mixed Capacitated Arc Routing Problem with three one-way streets and a two-way street requiring each of its sides to be serviced separately.

2.2.2 Intermediate facilities

The second extension of the classical CARP sees the introduction of multiple dumpsites. The extension is referred to as the Capacitated Arc Routing Problem with Intermediate Facilities (CARPIF) and was first studied by Ghiani et al. [32]. With the CARPIF a waste collection vehicle can unload its cargo at a nearest Intermediate Facility (IF). The vehicle can then resume its collection route. A route can thus exceed a vehicle's capacity, as long as the route contains visits to IFs, and the demand of the subroute between the IF visits does not exceed vehicle capacity.

The impact of an IF on waste collection routing is illustrated in Figure 2.3. The example is the same as the MCARP example of Section 2.2.1, i.e., a mixed network. Additionally the example sees the introduction of an IF shown in Figure 2.3a. The corresponding graph representation is shown in Figure 2.3b. Vehicle capacity is still fixed at 25 kg, but the vehicles are allowed to unload their waste at the dumpsite. The example assumes that the depot vertex can also be used as a dumpsite, hence it is unnecessary for a vehicle to unload before returning to the depot. With the new intermediate facility at v_{14} the entire area can be serviced by a single vehicle route (Figure 2.3c).

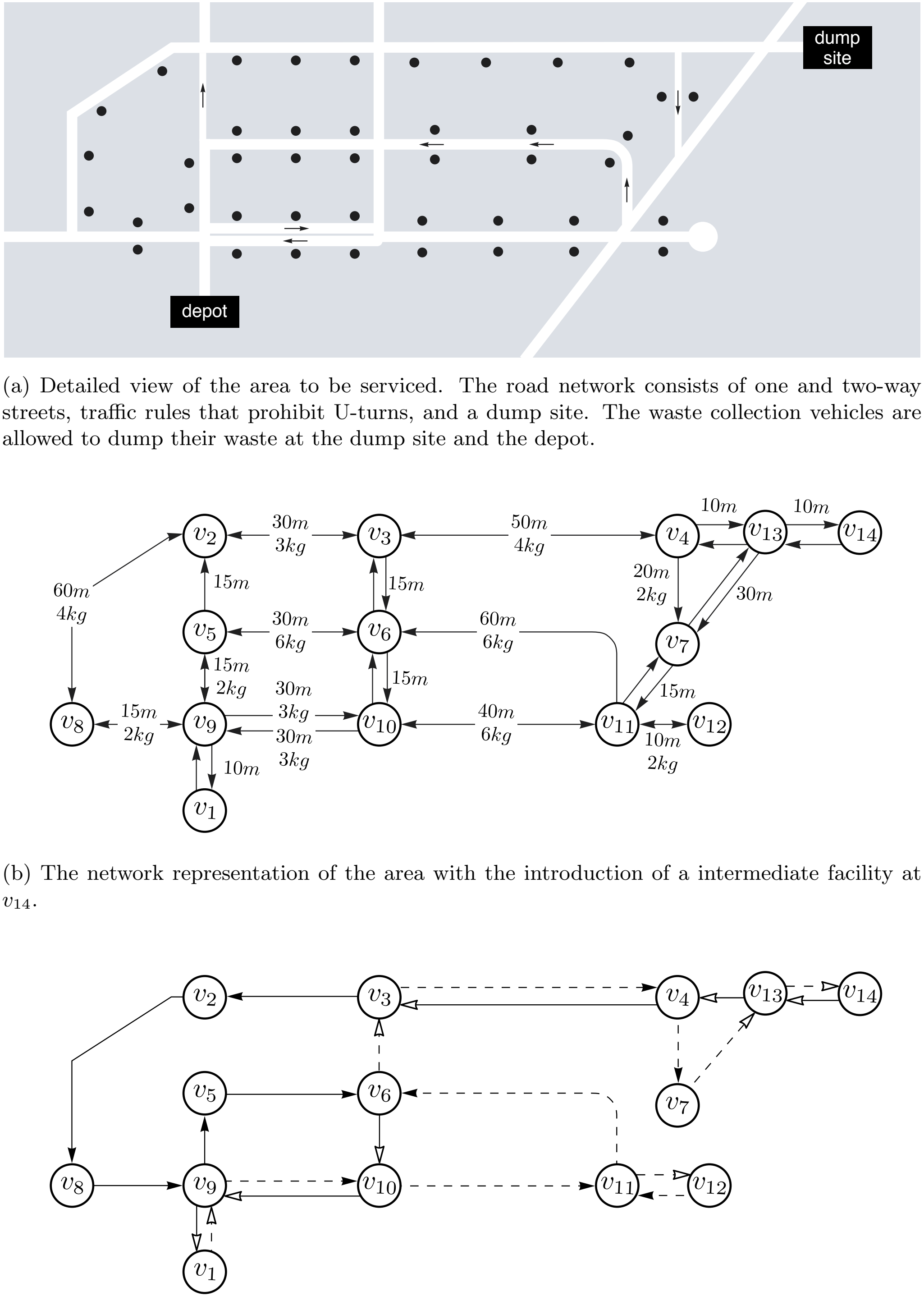
The CARPIF provides a closer representation of actual waste collection, but it still contains an unrealistic simplification. With the introduction of IFs a single vehicle can service any given area, even if the area contains thousands of arcs, as is the case with real street networks. Based on the work of Ghiani et al. [32], Ghiani et al. [31] study a more realistic extension of the CARPIF called the Capacitated Arc Routing Problem under Time Restrictions with Intermediate Facilities (CARPTIF), which is also studied in [33, 68]. With the extension a time restriction, independent from capacity, is placed on the duration of each vehicle's route—the restriction would typically be the number of working hours in a day. With the CARPTIF multiple vehicles are again required to service a given area, as is the case in reality.

The CARPTIF can be generalised to consider a mixed network. This version, termed the MCARPTIF, incorporates all the extensions considered in this section and is consistent with the solid waste collection problem as defined by Ghiani et al. [35]. The only remaining factor to consider for the formulation is the treatment of the vehicle fleet size.

2.2.3 Treatment of vehicle fleet size

In CARP literature, the fleet size is either fixed, assigned an upper bound value, left as a decision variable, or treated as unlimited. In most waste collection applications, see for example [4, 23, 30, 60, 61, 75, 76], and consistent with the original formulation of the CARP [39], K is an input value, usually corresponding to the current fleet size and it is treated as an upper bound; a solution with more than K routes is considered infeasible. Conversely, studies on CARP heuristics typically treat K as either unlimited [3, 12] or as a decision variable [6, 30, 33, 54]. In fact, Lacomme et al. [51] consider a limited fleet size to be an extension of the CARP. In studies where K is a decision variable, it is not minimised. The only objective is to minimise total route cost, and the resulting number of routes determines the fleet size requirements. Exceptions include the work of Chu et al. [15] who deal with a Periodic CARP in which K can be fixed or minimised, and the work of Grandinetti et al. [41] on a multi-objective CARP where one of the three objectives is to minimise K.

It can be argued that the objectives of minimising cost and the fleet size are not in conflict; thus solution approaches need only focus on the former. Additional routes result in additional dead-heading time to and from depots and IFs, as well as offload time. Since



(a) Detailed view of the area to be serviced. The road network consists of one and two-way streets, traffic rules that prohibit U-turns, and a dump site. The waste collection vehicles are allowed to dump their waste at the dump site and the depot.

(b) The network representation of the area with the introduction of a intermediate facility at v_{14}.

(c) With the introduction of the intermediate facility a single vehicle can now service the complete area. The complete route is [1, 9, 10, 11, 12, 11, 6, 3, 4, 7, 13, **14**, 13, 4, 3, 2, 8, 9, 5, 6, 10, 9, **1**]. Visits to intermediate facilities at v_{14} and v_1 are shown in bold. The total waste collected with the dashed subroute (v_1 to v_{14}) and the solid subroute (v_{14} to v_1) are 23 kg and 20 kg, respectively, and the total distance of the route is 520 meters. A solid arrow-head indicates that a vehicle services that edge or arc, and an empty arrow-head indicates that a vehicle only traverses it.

Figure 2.3: An example of the Mixed Capacitated Arc Routing Problem with Intermediate Facilities.

studies on the MCARP and CARPTIF do not report on the required K for the best cost solutions, it is unknown if existing solution methods for the problems can sufficiently deal with cases where K has an upper bound. This issue was identified by Belenguer and Benavent [7] for the CARP, who found that the number of vehicles required by heuristic solutions are not always equal to the possible minimum. Part of the aim of this thesis is to determine whether the same is true for the MCARPTIF.

2.2.4 The MCARPTIF formally defined

Based on the MCARP and CARPTIF reviewed in this section, and the potential treatments of vehicle fleet size, the MCARPTIF is formally defined as follows. The MCARPTIF considers a mixed 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 requiring each side to be serviced separately are modelled as two opposing arcs. 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_1. Each vehicle route must start and end at the depot and the sum of demand on a route may not exceed Q. The number of routes, K, can be minimised or must be less than the number of available vehicles, K_{UB}, and $K_{\text{UB}} \leftarrow \infty$ when an unlimited number of vehicles is available. For the MCARPTIF, vehicles are allowed to unload their waste at any IF and resume their collection routes. The duration of the offload is given as λ. IFs are modelled in $\boldsymbol{G}$ as $\boldsymbol{\Gamma}$, where $\boldsymbol{\Gamma} \subset \boldsymbol{V}$. The sum of demand on each subtrip between IF visits may not exceed Q, and unless $v_1 \in \boldsymbol{\Gamma}$, each vehicle has to visit an IF before returning to the depot. Lastly, a time restriction, L, is imposed on the duration of each vehicle route. There are two versions of the problem, depending on the treatment of K. For the first version the objective is to find a set of K feasible vehicle routes, servicing all required edges and arcs, that pre-emptively minimises total cost then the vehicle fleet size. The objective of the second version, which is the main focus of this thesis, is to find a set of K feasible vehicle routes that service all required edges and arcs, and pre-emptively minimises the vehicle fleet size, then total cost. When $\boldsymbol{A}$ is an empty set, denoted $\boldsymbol{A} = \varnothing$, the problem reduces to the CARPTIF; when $\boldsymbol{\Gamma} = \{v_1\}$ and $L \leftarrow \infty$, the problem reduces to the MCARP; and when both sets of conditions hold, the problem reduces to the CARP. MCARPTIF solution approaches can thus be used to solve all four problems. s Consistent with the work of Belenguer et al. [6], Lacomme et al. [51], and Mourão et al. [62], the mixed graph $\boldsymbol{G}$ can be transformed into a fully directed graph, $\boldsymbol{G}^* = (\boldsymbol{V}, \boldsymbol{A}^*)$, by replacing each edge, $(v_i, v_j) \in \boldsymbol{E}$, with two opposing arcs, $\{(v_i, v_j), (v_j, v_i)\} \in \boldsymbol{A}^*$. Arcs in $\boldsymbol{A}^*$ are identified by indices from 1 to $|\boldsymbol{A}^*|$. Lastly, each arc u has a dead-heading time, $d(u)$, denoting the time of traversing the arc without servicing it, and the total duration of any route cannot exceed the maximum allowed route time restriction, L.

The required arcs, $\boldsymbol{A}_r$, and edges, $\boldsymbol{E}_r$, in $\boldsymbol{G}$ correspond to a subset $\boldsymbol{R} \subseteq \boldsymbol{A}^*$ of required arcs such that $|\boldsymbol{R}| = 2|\boldsymbol{E}_r| + |\boldsymbol{A}_r|$. Each arc, $u \in \boldsymbol{R}$, has a demand, $q(u)$, a collection time, $w(u)$, and a pointer, $inv(u)$. 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 $inv(v) = u$. An arc task, u, represents an edge if $inv(u) \neq 0$. Furthermore, $q(u) = q(v)$, $d(u) = d(v)$ and $w(u) = w(v)$ for an edge's two opposing arcs, u and v. The depot is modelled by including in $\boldsymbol{A}^*$ a dummy arc, $\sigma = (v_1, v_1)$. IFs are also modelled in $\boldsymbol{A}^*$ as a set of dummy arcs $\boldsymbol{I} = \{\Phi_1, \ldots, \Phi_{|\boldsymbol{\Gamma}|}\}$, where $\Phi_i = (v_i, v_i)\ \forall\ v_i \in \boldsymbol{\Gamma}$, and $\boldsymbol{I} \subset \boldsymbol{A}^*$. All dummy arcs, including σ, have the following properties:

$$\left.\begin{aligned} inv(u) &= u \\ d(u) &= 0 \\ w(u) &= 0 \\ q(u) &= 0 \end{aligned}\right\} \forall\ u \in \boldsymbol{I} \cup \{\sigma\}. \tag{2.1}$$

An MCARPTIF solution, $\boldsymbol{T}$, is a list, $[\boldsymbol{T}_1, \ldots, \boldsymbol{T}_K]$, of K 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 list of arc tasks $[T_{i,j,1}, \ldots, T_{i,j,|T_{i,j}|}]$. The travel time for the shortest path from arc u to arc v, which excludes the time of dead-heading 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 the *Floyd-Warshall* algorithm that can incorporate forbidden turns and turn-penalties. A description of the algorithm is given in Section 2.A at the end of the chapter. Dijkstra's algorithm can also be modified for the shortest path calculations and can also incorporate forbidden turns and turn-penalties; we refer the reader to [51, 76] for implementation details.

The first subtrip in a route, $\boldsymbol{T}_{i,1}$, starts at the depot, and the last subtrip, $\boldsymbol{T}_{i,|\boldsymbol{T}_i|}$, ends with an IF and depot visit. All other subtrips start and end with IF visits while taking care that the starting IF of a subtrip coincides with the end IF of the previous subtrip. It is assumed that the shortest path is always followed between consecutive tasks. For the MCARPTIF, the best IF to visit, $\Phi^*(u, v)$, after servicing arc u and before servicing arc v can be pre-calculated using Equation (2.2), and the duration of the visit, $\mu^*(u, v)$, excluding offloading time, is given by Equation (2.3):

$$\Phi^*(u, v) = \arg\min\{D(u, x) + D(x, v) : x \in \boldsymbol{I}\}, \tag{2.2}$$

$$\mu^*(u, v) = D\big(u, \Phi^*(u, v)\big) + D\big(\Phi^*(u, v), v\big). \tag{2.3}$$

The best IF-visit calculation differs from the calculation of Belenguer et al. [6] as it excludes offloading time, which instead we include when calculating subtrip and route costs. It is assumed that the best IF is always visited between two arcs and between an arc and the depot.

For the MCARPTIF the load of a subtrip, $load(\boldsymbol{T}_{i,j})$, which may not exceed Q, is calculated as follows:

$$load(\boldsymbol{T}_{i,j}) = \sum_{n=1}^{|\boldsymbol{T}_{i,j}|} q(T_{i,j,n}). \tag{2.4}$$

The cost of subtrip $\boldsymbol{T}_{ij}$ is calculated using Equation (2.5), and the cost of the route $\boldsymbol{T}_i$ using Equation (2.6):

$$Z_s(\boldsymbol{T}_{i,j}) = \sum_{n=1}^{|\boldsymbol{T}_{i,j}|-1} \Big(D(T_{i,j,n}, T_{i,j,n+1})\Big) + \sum_{n=1}^{|\boldsymbol{T}_{i,j}|} w(T_{i,j,n}) + \lambda \tag{2.5}$$

$$Z(\boldsymbol{T}_i) = \sum_{j=1}^{|\boldsymbol{T}_i|} Z_s(\boldsymbol{T}_{i,j}). \tag{2.6}$$

Even though routes contain dummy arcs, Equations (2.4) and (2.5) remain accurate as a result of Equation (2.1). The preceding notations and encoding scheme are used throughout this thesis and are summarised in Tables 2.1 and 2.2.

Table 2.1: Glossary of mathematical symbols.

Problem symbols		Arcs (u and v) related symbols	
$\boldsymbol{R}$	Set of required arcs	$d(u)$	Dead-heading time of u
K_{UB}	Fleet size limit	$q(u)$	Demand of u
Q	Vehicle capacity	$w(u)$	Service time of u
L	Route time-limit	$inv(u)$	Pointer to the opposing arc of u
σ	Depot dummy arc	$D(u,v)$	Shortest dead-heading path time from u to v
$\boldsymbol{\Gamma}$	Set of IF vertices	$\Phi^*(u,v)$	Best IF to visit between u and v (eq. (2.2))
$\boldsymbol{I}$	Set of IF dummy arcs	$\mu^*(u,v)$	Duration of best IF visit between u and v (eq. (2.3))
λ	Unload cost		

Table 2.2: MCARPTIF solution representation symbols.

$\boldsymbol{T}$	MCARPTIF solution
$K = \lvert\boldsymbol{T}\rvert$	Number of vehicles required for $\boldsymbol{T}$
$\boldsymbol{T}_i$	Route i
$\boldsymbol{T}_{i,j}$	Subtrip j of route $\boldsymbol{T}_i$
$T_{i,j,n}$	n^{th} arc task in subtrip $\boldsymbol{T}_{i,j}$
$load(\boldsymbol{T}_{i,j})$	Demand of subtrip $\boldsymbol{T}_{i,j}$ (eq. (2.4))
$Z_s(\boldsymbol{T}_{i,j})$	Cost of subtrip $\boldsymbol{T}_{i,j}$ (eq. (2.5))
$Z(\boldsymbol{T}_i)$	Cost of route $\boldsymbol{T}_i$ (eq. (2.6))

The first aim of this literature review was to identify the appropriate ARP formulation for the MCARPTIF. The second aim and focus of the next section is to identify appropriate solution approaches for the problem.

2.3 Solution approaches for Capacitated Arc Routing Problems

Despite what we consider to be a common practical application, the MCARPTIF has not been formally studied in literature. A close variant of the problem is studied by Santos et al. [76] which deals with a single IF, a heterogeneous fleet and with demand at intersections as well as road segments. Ghiani et al. [30] deal with an MCARPTIF with additional requirements such as a heterogeneous fleet where only certain vehicles can service certain street types. Rodrigues and Soeiro Ferreira [73] also deal with an MCARPTIF with a heterogeneous fleet and with landfills that only allow a limited number of visits. An extended MCARP is studied in [60, 61] that considers one IF but without a route duration limit. Instead, each available vehicle in the fleet must be used and each route must consist of at least two subtrips. Ghiani et al. [35] consider a real-world Mixed Arc Routing Problem under Time Restrictions where waste collection vehicles are only constrained by a route duration restriction, and not by vehicle capacity. Other CARP variants, not considered in this thesis, with application in waste collection include the CARP and MCARP with multiple instead of one depot [2, 50]; the Sectoring Arc Routing Problem [24, 62]; and the MCARP with non-overlapping routes [17]. We refer the reader to [35, 64] for a comprehensive review of relevant CARP variants.

Most ARP research focus on the CARP, with a few studies done on the MCARP and CARPTIF. The focus of this section is on reviewing solution approaches for the three problems, which can be extended to the MCARPTIF. Solution approaches are briefly evaluated[2] for the three execution time-limits defined in the previous chapter. Under short execution times a feasible solution has to be found in limited time. The second case deals with medium execution-times where more time is available to improve a solution. And the third case deals with longer execution times where more time is available to improve the solution even further. Before discussing the solution approaches we first review how difficult the MCARPTIF is to actually solve.

2.3.1 Problem difficulty

Problems that can be solved in polynomial time are typically solved to optimality using efficient algorithms and exact methods [90]. Problems that are usually considered hard to solve are those for which there are no known polynomial algorithms. Such problems are called nondeterministic $\mathscr{NP}$-class problems. Belonging to this class are $\mathscr{NP}$-hard problems where $\mathscr{NP}$-hard implies that the solution space of the problem will increase at an exponential or factorial (non-polynomial) rate as the number of vertices/arcs of the problem increases. Golden and Wong [39] show that the CARP, the simplest form of the MCARPTIF, belongs to this class of problems. This means that all of the extensions of the CARP, including our MCARPTIF, are also $\mathscr{NP}$-hard and difficult to solve to optimality using exact methods. As a result, the most effective methods for dealing with the problems are based on heuristic and metaheuristic solution techniques [19].

In literature, the standard approach when dealing with a new CARP variant is to simultaneously develop lower and upper bounds. Classical benchmark sets are then modified to account for new problem characteristics, and upper bounds, found using heuristics, are compared against good lower bounds and against optimal solution values for smaller test instances. Lower-bounding and exact approaches, reviewed in [1, 8], have been successfully applied to the CARP and MCARP, but as a result of their computational complexity, these approaches are not yet capable of dealing with realistically sized instances [57]. The largest CARP instance solved using an exact approach with a six hour execution time-limit consists of 159 required edges [3], and the largest MCARP solved within a one hour execution time-limit consists of 23 and 77 required arcs and edges, respectively [6, 40]. Real world waste collection problems, such as the case study instance of Bautista et al. [4], can contain thousands of edges. Also, lower bounding and exact methods have not been applied to the CARP with more than one IF. Just developing appropriate Integer Linear Programming formulations for these extensions is, on its own, a challenging research topic [14, 31]. As the first formal investigation on the MCARPTIF the focus of this thesis is only on heuristic methods for the problem. We leave the development of exact solution approaches, including lower bounds, for future work.

Heuristics attempt to find good feasible solutions to optimisation problems, within reasonable computing time, but without any guarantee that the solutions are optimal. For this thesis we distinguish between two types of heuristics: constructive heuristics that quickly generate initial solutions for the CARP and its extensions, as is required when execution time is limited; and improvement heuristics that, as their name implies, progressively improve the initial solutions. The improvement heuristics are computationally more involved than constructive heuristics, but they do return better solutions, making them ideal when more execution time is available.

[2] More detailed descriptions of existing methods are presented at the start of each chapter in which the MCARPTIF heuristics are developed.

2.3.2 Constructive heuristics

Constructive heuristics for CARPs remain an important area of research. As motivated by Santos et al. [77], these heuristics generally provide good solutions in acceptable CPU time which is an important criterion in many real waste collection applications. Execution times for advanced improvement heuristics can become large for even modestly sized problems, making them impractical for certain applications. Owing to their simplicity, constructive heuristics are flexible and can be more easily modified to extensions of the CARP, which makes them easier to implement, and they do not require the determination and fine tuning of parameters. They also form the starting point for improvement heuristics, see for instance [6, 12, 31, 63, 68]. Polacek et al. [68] show that for their CARP improvement heuristics, linking it with better quality initial solutions leads to better convergence and higher quality final solutions. Lastly, constructive heuristics are used to solve sub-problems of certain CARPs such as those studied in [41, 62].

Numerous constructive heuristics, which follow simple greedy rules to progressively build approximate feasible solutions, have been proposed for the CARP. Since the second aim of this review is to identify existing solution approaches that can be extended to the MCARPTIF we mainly focus on constructive heuristics that have been applied to the MCARP and CARPTIF. For more information on constructive heuristics for the classical CARP we refer the reader to [19, 69]. The heuristics that we considered include *Path-Scanning*, *Improved-Merge* and *Route-First-Cluster-Second* applied to the MCARP by Belenguer et al. [6]; and *Route-FirstCluster-Second-* applied to the CARPTIF by Ghiani et al. [31]. *Route-First-Second-Cluster* heuristics rely on splitting procedures that are also extensively used in certain advanced improvement heuristics.

The splitting procedures and constructive heuristics are reviewed in more detail in Chapters 3 and 4, in which we develop heuristics for the MCARPTIF and critically evaluate the heuristics on their ability to find feasible solutions within short execution times.

2.3.3 Improvement heuristics

The most popular improvement techniques for CARPs are based on Local Search (LS) methods [69]. Gendreau [29] summarises LS as an iterative procedure that, starting from an initial feasible solution, progressively improves it by applying a series of local modifications or moves. At each iteration the search moves to an improving feasible solution that differs only slightly from the current one, and the search terminates when it encounters a local optimum with respect to the transformation that it considers.

For CARPs, LS moves consist of simple modifications to the current solution that change the service position of required arcs or edges in a solution's routes. For instance, one type of move may consist of taking an arc serviced in a route out of its current position and placing it in a different route for service. The move set will then consist of the removal of each arc or edge from its current position and its placement in all other possible positions, where it does not result in a constraint violation. The best or first improving move may then be implemented whereby LS moves to the improved solution. The search then repeats from the improved solution and terminates when no improving moves can be found. Local optima at which LS terminates are often still fairly low quality solutions, which is not ideal when high-quality solutions are required.

To our knowledge, there are no CARP studies that exclusively use LS to improve initial solutions. Instead LS is used within more intelligent improvement strategies, referred to as metaheuristics, which are currently the most successful algorithms for the CARP [69]. Metaheuristics are general master strategies that try to avoid local optima either by intelligently manipulating the LS framework, or by considering several solutions at a time. With proper allowance for execution times, these advanced methods are capable of generating high quality solutions. For an overview of metaheuristics we refer the reader to [37, 80] and for an overview of their application to CARPs we refer the reader to [64, 69].

Metaheuristics for CARPs all rely on direct extensions of LS. Some of these applications include Variable Neighbourhood Descent [23, 45]; Variable Neighbourhood Search [68]; Tabu Search[2, 12, 31, 44]; Tabu Scatter Search [42]; and Guided Local Search [10, 63]. Other metaheuristics in which LS is embedded and directly called include Greedy Randomized Adaptive Search Procedure (GRASP) [83]; Ant Colony Optimization [33, 78]; and Memetic Algorithms (MA) [51, 54, 81]. The MA of [6] is the only metaheuristic that has been applied to the MCARP. The listed applications include all the recent metaheuristics for CARPs reviewed by Muyldermans and Pang [64] and Prins [69], and as mentioned all rely on some form of LS. Together with constructive heuristics to generate initial solutions, LS is thus an important building block in developing metaheuristics for the MCARPTIF.

Another importing building block is splitting procedures [70]. Many of the metaheuristics work on giant tours, which are partitioned into feasible vehicle routes via splitting procedures. Giant tours are used in MAs to encode chromosomes and in the Ant Colony Optimization of Santos et al. [78] to generate new solutions. They are also used in *Route-First-Cluster-Second* constructive heuristics for CARPs during the Cluster-Second phase [6, 31, 51, 62, 68].

With the emphasis of this thesis on solving large MCARPTIFs under different execution time-limits, efficient implementations of LS and of splitting procedures are critical. Inefficient implementations may prohibit metaheuristics to effectively deal with realistically sized instances. This issue is raised by Prins [69] who recommends two research directions for the CARP that should be investigated in priority, which we believe also applies to the MCARP and MCARPTIF. The first is to develop metaheuristics that are able to tackle in reasonable running times huge instances met in real applications, such as residential waste collection, which according to Prins [69] requires the service of 10000 arcs and edges, or more. Currently only constructive heuristics are capable of dealing with these sized instances. The second priority, which we do not address in this thesis, is to exploit parallel or multi-core computers in solving CARPs.

Efficient splitting procedures are developed by Lacomme et al. [51] for the CARP, which are applied as-is on the MCARP in [6, 62]. Ghiani et al. [31] develop a splitting procedure for the CARPTIF that can also be used as-is on the MCARPTIF, but its efficiency has not been formally tested. In response, Chapter 3 is dedicated to splitting procedures for the MCARPTIF in which we develop efficient procedures that outperform the version of Ghiani et al. [31]. Next, in Chapter 4 we develop constructive heuristics for the MCARPTIF, some of which employ the splitting procedures. The efficiency of LS is seldom critically evaluated in CARP literature, and is thus the subject of Chapters 5 and 6 in which we develop efficient LS procedures for the MCARPTIF. In Chapter 7, the LS procedures are extended to develop a Tabu Search metaheuristics for the MCARPTIF, which we further test as-is on the MCARP and compare against the Memetic Algorithm of Belenguer et al. [6]. All the heuristics are critically evaluated on their ability to find high-quality solutions under short, medium and long execution times.

2.4 Evaluating heuristics

The main limitation of approximate solution techniques is that there is no guarantee that they will produce optimal solutions. In fact, it is difficult to prove that the generated solutions are even close to optimal. This makes solution evaluation and validation difficult, yet critical. This section reviews two components central to evaluating heuristics. First is the evaluation criteria used to measure the performance of heuristics, second is the test instances on which heuristics are tested and on which the performance measurements are captured.

2.4.1 Evaluation criteria

In CARP studies, heuristic performance is evaluated on two criteria: solution quality, usually in the form of total route costs, and algorithm efficiency, which is measured as the CPU time required by the algorithm to find or improve on a feasible solution. Algorithm efficiency can also be evaluated by determining the order of growth of the algorithm's running time, as given by the commonly used big-$\mathcal{O}$ notation. More information on the notation can be found in Section 2.B at the end of the chapter.

When possible, solution quality is measured by comparing heuristic results to optimal values found on small instances and on lower bounds for larger instances. Tight bounds have been developed for the CARP [3, 11], and recently for the MCARP [6, 40]. Ghiani et al. [30] calculate weak lower bounds for the CARPTIF with one IF by using the algorithm of De Rosa et al. [22]. None exist for the CARPTIF with more IFs, nor for the MCARPTIF, and their development falls beyond the scope of this thesis.

When lower bounds are not available, an option is to substitute them with the best solutions found during all the computational tests [72, 80], which serves our test purpose as it gives a general quality measure between heuristics. For this thesis, we also measured solution quality in terms of the required number of vehicles, K. A tight lower bound on the optimal number of vehicles, K_{LB}, for a CARP and MCARP solution can be calculated using Equation (2.7):

$$K_{\text{LB}} = \left\lceil \frac{\zeta}{Q} \right\rceil, \tag{2.7}$$

where ζ is the total demand of the required arcs and edges of the problem instance, and Q is vehicle capacity. For the MCARPTIF, Equation (2.7) only gives a lower bound on the number of subtrips with IF visits, since route duration is constrained by a time-limit, not capacity. The K_{LB} can instead be calculated using Equation (2.8)

$$K_{\text{LB}} = \left\lceil \frac{O + \lambda \left\lceil \frac{\zeta}{Q} \right\rceil}{L} \right\rceil, \tag{2.8}$$

where O is the total service time of the required arcs and edges of the problem, λ is the IF offloading time and L is the route duration restriction. The dividend of Equation (2.8) is a weak lower bound on solution cost as it ignores dead-heading[3] times. As a result, the fleet size lower bound is also weak. Finding tight lower bounds for the MCARPTIF is not trivial and one option would be to divide good lower bound cost values by L. Otherwise, similar to total cost, solutions can be compared against best solutions found during all computational tests, which is the approach used in this thesis.

[3] Dead-heading is the travel between required arcs, edges, depots and IFs when no waste collection takes place.

2.4.2 Problem test instances

When evaluating and comparing heuristics, the choice of test instances is critical as their structure can influence heuristics performance. Real-life instances, such as those solved in [4, 23, 30, 75, 76], constitute good benchmarks to carry out performance evaluations, but are not openly available. As a result, studies on CARPs have relied on randomly constructed instances. Over time these instances have become standard classical benchmark sets used to evaluate CARP solution methods.

As cautioned by Rardin and Uzsoy [72], classical benchmark sets introduce subtle biases that need to be recognised. First, the posted instances may not model real world environments. In fact, some of the instances may not be intended to be representative of applications at all. They are merely used to validate that algorithms are functional. Second, results are published for methods that perform well on the classic sets, which may introduce a hidden bias against alternative algorithms that perform poorly on the classic sets, but may perform well on realistic instances. Lastly, the wide use of the benchmark sets may result in researchers spending too much effort on making algorithms perform well on specific instances, at the risk of the algorithms becoming fragile and performing poorly on other instances not included in the classical sets.

In an effort to identify these biases the next section critically evaluates existing benchmark sets for CARPs. Furthermore, in an effort to overcome the biases we have developed new MCARPTIF benchmark sets, representative of actual waste collection. These sets are introduced and compared against the existing sets at the end of the section.

Existing benchmark sets

Some of the first CARP benchmark sets are the *gdb* and *bccm*[4] sets introduced by Golden et al. [38] and Benavent et al. [9], respectively. The sets were mostly generated by hand [7] and do not closely model any real environment. Their purpose was to demonstrate that heuristics were capable of producing feasible solutions for the (then) new problems. Since all *gdb* instances have been solved to optimality, Lacomme et al. [51] and Corberán and Prins [19] state that they should no longer be used to compare CARP metaheuristics, and between Bartolini et al. [3] and Bode and Irnich [11], the *val* instances are also solved to optimality. Because of their small size, heuristics and metaheuristics scalability issues may also go unnoticed on these instances [59]. Despite their short-comings, both sets are still used to test and rank heuristics and metaheuristics; see for instance [54, 58, 68, 77, 83].

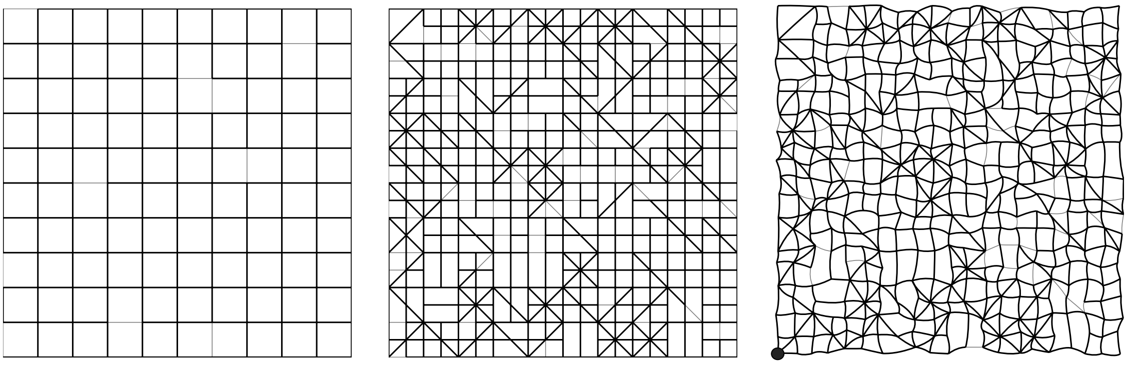
Two benchmark sets, which are adapted from the *gdb* and *bccm* have been proposed for the CARPTIF by Ghiani et al. [31]. These instances, which we refer to as *gdb-IF* and *bccm-IF*, are solved in Ghiani et al. [31, 33] and in Polacek et al. [68]. Unfortunately, the original instances used by Ghiani et al. [31], and solved in Ghiani et al. [33] are no longer available, and applying the transformation process documented by the authors results in inconsistencies. The only CARPTIF results available in literature that can be used for validation and comparison of new approaches are those reported by Polacek et al. [68] on the *bccm-IF* instances, but the set has the same shortcomings of the original *bccm* set. Furtermore, the original *bccm* instances had separate deadheading and service times per edge. Since the service times are fixed for any feasible CARP solution, Belenguer and Benavent [7] removed it from the instances and worked with only the deadheading times. The reduced versions have been predominantly used in literature since. For the CARPTIF, service time plays a role due to the route time-limit. Results reported in [31] indicate that the CARPTIF transformation was performed on the original *bccm* instances, with service

[4] The *bccm* set is also referred to as the *val* set.

times. Polacek et al. [68] performed tests on the transformed *bccm* instances without service times, and the authors incorrectly compare their metaheuristic results directly against those of Ghiani et al. [31]. We also found inconsistencies when transforming and solving *gdb* instances, which we reported in [87].

Recent CARP studies have relied more on the *eglese* [3], *egl-large* [57] and *bmcv* [10] set, which are all based on actual winter-gritting applications in which salt is to be spread on icy roads. Eighty-six out of the hundred *bmcv* instances have been solved to optimality by Bartolini et al. [3] and Bode and Irnich [11], limiting their usefulness, but only eleven of the twenty-four *egl* instances and none of the ten *egl-large* instances have been optimally solved, thus rendering them good candidates for evaluating heuristics. Both, however, are undirected and based on snow removal.

Belenguer et al. [6] generated fifteen random *lpr* instances on mixed graphs that mimic waste collection. Figure 2.4 shows the resulting road network when using the approach, as documented by Belenguer et al. [6], to create a problem instance similar to *lpr-c-05*, which is the largest instance in the set. Belenguer et al. [6] also create 34 *mval* instances with mixed graphs derived from the *bccm* set. The *lpr* and *mval* sets are currently the only available MCARP benchmarks. The CARP sets are publicly available from `http://logistik.bwl.uni-mainz.de/benchmarks.php` and the MCARP sets from `http://www.uv.es/belengue/mcarp/`.



(a) Grid is constructed that randomly consisting of arcs (thin-lines) and edges (thick-lines)

(b) Arcs and edges are randomly removed from the grid and new ones are randomly inserted.

(c) Network is distorted by randomly moving vertices and making arcs and edges curved.

Figure 2.4: Randomly generated *lpr-c-05* like network

New benchmark sets

Since no benchmarks exist for the MCARPTIF, we developed seven benchmark sets for the problem. Five are based on existing CARP and MCARP benchmarks, and the remaining two are new and based on real road networks on which waste collection occurs. A full description of the benchmark sets is available from Willemse and Joubert [86].

For the first new set the area shown in Figure 2.5 was used to develop three *Act-IF* instances. To generate the instances, the known total amount of weekly waste generated in the study area was evenly distributed among its households. Edge demands for the network were then calculated using the number of households on each road segment multiplied by the waste generated per household. Dead-heading travel speed was taken as 28 km/h, and service time per edge was calculated using a collection time of 1 second per kilogram of waste plus a travel speed between bins of 14 km/h. We assumed it takes 5 minutes for

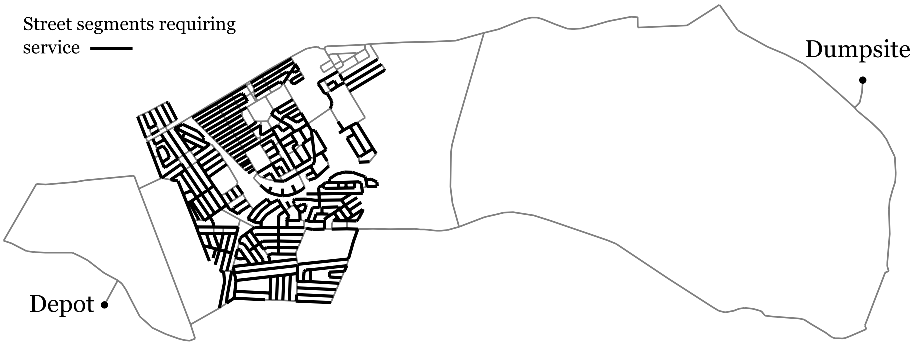


Figure 2.5: *Act-IF* road network

a vehicle to offload its waste at an IF, and vehicle capacity was set to 10000 kg. Waste collection crews operated from 08:00 AM to 17:00 PM, including breaks totalling 1 hour, typically taken when the vehicle visits the dumpsite. The route time-limit was accordingly set to 8 hours. All instances have one IF, coinciding with the dumpsite, situated outside the study area and away from the vehicle depot. The instances are relatively small, ranging in size from 151 to 400 required edges.

The second set consists of three *Cen-IF* instances that are based on actual road networks of Centurion, which forms part of the City of Thswane Metropolitan Municipality, South Africa. The benchmark set was created by dividing the Centurion area as shown in Figure 2.6 into three sub-areas, and the instances range in size from 1012 to 2755 required arcs and edges, making them some of the largest benchmarks currently available. The procedure used to develop the benchmark can be found in Section 2.C at the end of the chapter. Two of the instances, *Cen-IF-a* and *Cen-IF-c*, have one IF that coincides with the vehicle depot, and are thus not strictly MCARPTIFs but Mixed Capacitated Arc Routing Problems under Time restrictions (MCARPTs). *Cen-IF-b* has two IFs, one coinciding

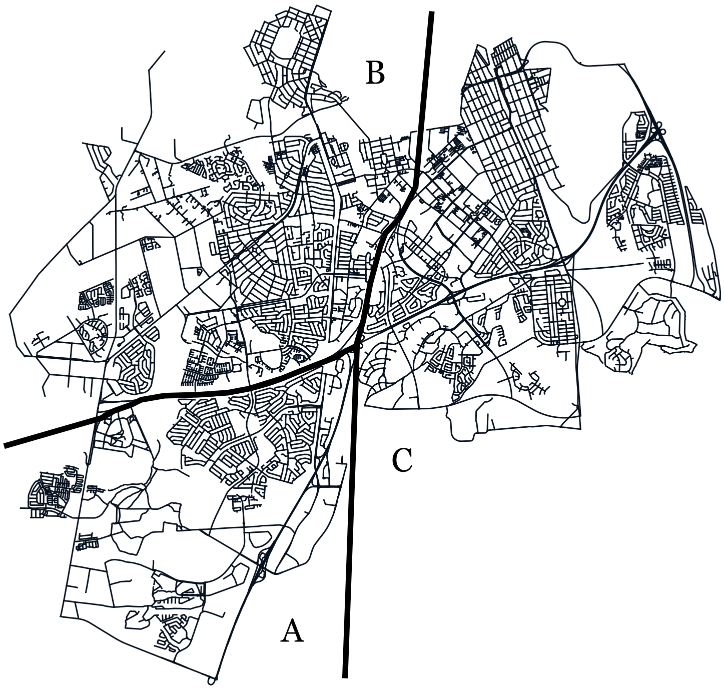


Figure 2.6: *Cen-IF* road network

with the vehicle depot. A fourth benchmark instance was also created, *Cen-IF-Full*, that consists of the entire area. The problem has 6289 required arcs and edges, making it the largest ARP test instance currently available. The instance was exclusively used for our final computational tests in Chapter 7.

The third MCARPTIF set that we developed was based on the existing *lpr* instances. We included IFs at vertices $\lfloor |\boldsymbol{V}|/2 \rfloor$ and $2\lfloor |\boldsymbol{V}|/2 \rfloor$, and included a route duration time restriction of 28800 seconds. The set, referred to as *Lpr-IF*, is especially useful during early heuristic development, or when studying new CARP versions, since it contains a wide range of instances.

Since the original and transformed instances of Ghiani et al. [31] are no longer available, we treated the *gdb-IF* set generated using their approach as a new CARPTIF set. We also transformed and solved available *bccm* instances, thus rendering our results comparable to Polacek et al. [68]. The *gdb-IF* and *bccm-IF* instances have a low number of arcs per subtrip ratio. As a result, the capacity limit is rarely reached on these instances, meaning that most routes will not have intra-route IF visits. Two new sets, *gdb-IF-3L* and *bccm-IF-3L*, were developed specifically to overcome this issue. To develop the sets the *gdb-IF* and *bccm-IF* were used as-is except for the route duration limit that was increased by a factor of three for each problem instance. A new *mval-IF-3L* set was also developed based on the *mval* set. The same transformation used to develop the *gdb-IF* was applied and the route duration limits of *bccm-IF-3L* were imposed.

All the CARPTIF and MCARPTIF benchmark sets used in this thesis are publicly available from Willemse and Joubert [88].

Benchmark set comparison

Key features of specific waste collection benchmark instances and all benchmarks sets are shown in Table 2.3. For comparison, the table shows features of the six *Act-IF* and *Cen-IF* instances, the three largest *Lpr-IF* instances, and the two case study problem instances of Bautista et al. [4] and Ghiani et al. [30]. Starting with the *Act-IF* set, its key features are consistent with the case study instance of Ghiani et al. [30]. *Act-IF* instances are smaller than the instance of Bautista et al. [4], but it should be noted that *Act-IF* set only represents a small suburb of the total municipal area that requires service. The *Cen-IF* instances are much larger than the two case study problem instances in [4, 30], which may detract from the realism of the set. In practice, routes may only have to be developed for smaller sub-areas, or sectors, linked with collection days. This makes the set ideal for tests on MCARP sectoring type problems studied in [24, 62]. Comparing *Act-IF* and *Cen-IF* to the case study instances, their arcs per route ratios are slightly higher than the instance of Ghiani et al. [30] but lower than the instance of Bautista et al. [4]. As for the *Lpr-IF* set, features of the larger instances are consistent with the case study problem of Bautista et al. [4], and features on smaller instances are consistent with *Act-IF*. Overall, the new *Act-IF*, *Cen-IF* and *Lpr-IF* sets have features that arc consistent with either the Bautista et al. [4] or Ghiani et al. [30] instances, which confirms that they are representative of actual waste collection.

As shown in Table 2.3, the *gdb-IF* and *bccm-IF* instances have a low number of arcs per subtrip ratio. The number of arcs per route are the same or even lower than the arcs per subtrip. Furthermore, the fleet size lower bound calculation used is weak, meaning the actual number of arcs per route is even lower than indicated. As a result, and confirmed during our computational tests, the capacity limit is rarely reached on these instances, meaning that most routes for the instances do not have intra-route IF visits. Compared to waste collection sets, the *gdb-IF* and *bccm-IF* sets are small and all the edges require

Table 2.3: Benchmark set and problem instance features.

Benchmark sets			Subtrips		Routes	
and instances	$\lvert \boldsymbol{A}' \cup \boldsymbol{E}' \rvert$ (1)	$\lvert \boldsymbol{A}_r \cup \boldsymbol{E}_r \rvert$ (2)	min # (3)	#Arcs/subtrip (4)	K_{LB} (5)	#Arcs/route (6)
Waste collection problem instances						
Case study of [4]	540	679	5	136	NA	NA
Case study of [30]	223	153	3	51	NA	NA
Act-IF-a	259	401	2	76	1	151
Act-IF-b	509	151	6	67	2	201
Act-IF-c	410	250	4	63	2	125
Cen-IF-a	441	1012	17	60	7	145
Cen-IF-c	486	2519	37	68	15	168
Cen-IF-b	360	2755	39	71	16	172
Cen-IF-full	1282	6289	92	68	41	153
Lpr-IF-a-05	250	806	18	45	4	76
Lpr-IF-b-05	75	801	18	45	6	84
Lpr-IF-c-05	38	803	23	35	9	89
Mean values over randomly generated benchmark sets						
Lpr-IF	71	352	7	28	3	57
mval-IF	0	88	5	21	3	32
gdb-IF	0	29	6	5	6	5
gdb-IF-3L	0	29	6	5	2	14
bccm-IF	0	63	5	15	6	10
bccm-IF-3L	0	63	5	15	2	31
Mean values over realistic snow-removal benchmark sets						
bmcv [10]	22	66	6	12	-	-
eglese [53]	34	110	15	7	-	-
egl-large [12]	14	361	31	12	-	-

(1): number of arcs and edges not requiring service; (2): number of arcs and edges requiring service; (3): lower bound on minimum number of subtrips; (4): (2) ÷ (3); (5): weak lower bound on fleet size; (6): (2) ÷ (5); NA: Not enough information available to calculate (5) and (6).

service. The random *gdb-IF* set has very few required edges and has low arcs per route values, but not that much less than *eglese* winter-gritting instances. The *bccm-IF* and *mval-IF* set is also consistent with winter-gritting, being very similar to *bmcv*. The original *mval*, *bccm* and *gdb* sets are thus more representative of winter-gritting than waste collection.

There are two main differences between waste collection instances, including our new instances and the case study instances [4, 30], and the winter-gritting sets of Beullens et al. [10], [12] and Li and Eglese [53]. First, the ratio of non-required to required arcs and edges is higher for waste collection sets. Only arcs and edges with residential properties accumulate waste and require service. The winter-gritting networks only contain major roads, most of which require service. Second, the number of arcs per route is much higher for waste collection. This is due to waste collection typically taking place in urban and city areas with relatively short road-segments.

We chose not to modify the *bmcv*, *eglese* and *egl-large* sets into CARPTIF instances for two reasons. First, instance data of the sets are not time-based, making it difficult to assign realistic route duration limits to the sets. Second, as shown in Table 2.3 waste collection sets have different characteristics to winter-gritting sets. A key question is whether the problem instance characteristics influence heuristic performance. If so, the results on waste collection sets have to be prioritised for results to be of value to waste collection planners. This effect has not been formally investigated in literature, and such a detailed comparative study is not included in the scope of this thesis. To ensure that our results are of practical value computational experiments are predominantly performed on the *Cen-IF*, *Act-IF* and *Lpr-IF* waste collection sets. We do test the effect that the *gdb* and *bccm* based sets have on constructive heuristic performance, but only to a limited extent.

2.5 Conclusion

Waste collection routing is a difficult and complex optimisation problem, too complex to be formulated as a pure CARP. In this chapter we have formally defined the MCARPTIF which accurately models residential waste collection. We briefly reviewed constructive heuristics that are capable of quickly generating solutions for CARPs when execution time is limited. Improvement heuristics and more advanced metaheuristics were also briefly reviewed. These methods can be used to improve solutions of constructive heuristics when more execution time is available. The remainder of this thesis will focus on developing and extending the identified heuristics to deal with the MCARPTIF, and on critically evaluating the heuristics on realistic waste collection benchmark problems.

Some of the constructive heuristics rely on giant tour based approaches that are also used in metaheuristics. These procedures are reviewed in more detail in the next chapter in which we develop and test new giant tour procedures for the MCARPTIF. In Chapter 4, existing constructive heuristics are reviewed and new heuristics are developed. Existing LS procedures for the MCARP are reviewed in Chapter 5 and adapted to the MCARPTIF. In Chapter 6, advanced LS acceleration mechanisms are developed and tested for the problem. The accelerated LS procedures can be used when execution time is limited to improve constructive heuristic solutions. LS is an important component of the metaheuristic that we developed in Chapter 7. The metaheurstic, which can be used when more execution time is available, is critically evaluated on the large MCARPTIF instances.

Chapter appendix

2.A Modelling and calculating shortest paths with turn penalties

To model turn penalties on the directed graph $\boldsymbol{A}^*$, let $b(u)$ be the beginning vertex of arc u and let $e(u)$ be the end vertex of the arc. Arcs u and v are said to be directly connected if $e(u) = b(v)$ or $e(v) = b(u)$. We then let $suc(u)$ be the set of allowed successor-arcs of u, i.e., $v \in suc(u)$ if $e(u) = b(v)$ and the turn from u to v is allowed. Let, $pen(u, v)$ be the cost of turning from arc u into arc $v \in suc(u)$. Given arcs u and v we define a feasible deadheading path from u to v as a sequence of arcs $\boldsymbol{P} = [u = u_1, u_2, \ldots, u_k = v]$, such that

$$u_{i+1} \in suc(u_i) \qquad \forall \ i = \{1, \ldots, k-1\}. \tag{2.9}$$

The deadheading cost for the path, $C(\boldsymbol{P})$, is defined by

$$C(\boldsymbol{P}) = pen(u_1, u_2) + \sum_{i=2}^{k-1} \big(d(u_i) + pen(u_i, u_{i+1})\big). \tag{2.10}$$

Because we only include required arcs in our solution encoding scheme, the costs of deadheading arcs u and v are excluded with Equation (2.10).

Similar to the adaptation of Dijkstra's algorithm in [51], we have adapted the *Floyd-Warshall* algorithm for a directed graph as shown in Algorithm 2.1 to pre-compute a shortest path between all pairs of arcs while accounting for turn penalties. The cost of the shortest path from arc u to arc v is given by $D(u, v)$, and the predecessor of v on this path is given by $\Pi(u, v)$. Paths to and from the depot and IFs are handled since they are modelled in $\boldsymbol{A}^*$ as dummy arcs.

The algorithm starts by setting the distance from an arc to each of its successor arcs equal to the turn penalty between the two; the distance from an arc to itself is set equal to zero; and the distances between each arc and its non-successor arcs are initially set to infinity. Next the algorithm determines if a path from arc u to arc v via an intermediate shortest path to arc k is shorter than the current shortest path between u and v. If so, the new shortest path between u and v is changed to include a visit to k, and the predecessor arc function Π is updated accordingly. This phase is repeated for all $u, v, k \in \boldsymbol{A}^*$. The arcs in $\boldsymbol{A}^*$ are assigned to u, v and k in the same order, hence the final shortest paths between $u - k$ and $k - v$ are determined before comparing $u - k - v$ to $u - v$. Finally, using Π as input, the shortest arc path between arcs u and v is determined with Algorithm 2.2. Both D and Π are pre-computed and provided as input data for our algorithms.

Algorithm 2.1: *Arc-to-Arc-Floyd-Warshall* algorithm with turn penalties

```
Input  : $\boldsymbol{A}^*$, $d(u)$, $suc(u)$ and $pen(u,v)$ where $u, v \in \boldsymbol{A}^*$.
Output: $D$ and $\Pi$.

for $u \in A^*$ do
    for $v \in A^*$ do
        if $u = v$ then
            $D(u,v) = 0$ and $\Pi(u,v) = 0$;
        else if $v \in suc(u)$ then
            $D(u,v) = pen(u,v)$ and $\Pi(u,v) = u$;
        else $D(u,v) \leftarrow \infty$;

for $k \in A^*$ do
    for $i \in A^*$ do
        for $j \in A^*$ do
            if $D(i,k) + D(k,j) + d(k) < D(i,j)$ then
                $D(i,j) = D(i,k) + D(k,j) + d(k)$;
                $\Pi(i,j) = \Pi(k,j)$

return $(D,\ \Pi)$
```

Algorithm 2.2: *Shortest-Path-Arc-Sequence*

```
Input  : $\Pi$, origin arc $u$ and destination arc $v$.
Output: $\boldsymbol{P} = [u, a_i, a_j, \dots, v]$.

$\boldsymbol{P} \leftarrow [v]$;
$a \leftarrow v$;
while $a \neq u$ do
    $a = \Pi(u,a)$;
    insert $a$ in the beginning of $\boldsymbol{P}$
return $(\boldsymbol{P})$
```

2.B Generic algorithm running time analysis

A simple and generic way to evaluate an optimisation algorithm is to determine the order of growth of its running time, defined as a growth function. As stated by Cormen et al. [20], the growth function gives a simple characterisation of an algorithm's efficiency and may also be used to compare the relative performance of alternative algorithms on the same problem.

In spirit of Cormen et al. [20] we employ an asymptotic notation to describe the running time of our optimisation algorithms, specifically focusing on the worst-case running-time of a function $T(n)$. The function is defined on integer input sizes, which for our implementations correspond to the number of required arcs of the problem network, or the number of arcs in a subset of the required arcs of a network. Specifically we employ $\mathcal{O}$-notation, which is an asymptotic upper bound of the running time function $T(n)$. For a given function $g(n)$ we denote by $\mathcal{O}(g(n))$ the set of functions

$$\begin{aligned}\mathcal{O}(g(n)) = \{g(n) : &\text{ there exist positive constants } c \text{ and } n_0 \text{ such that} \\ &\mathcal{O} \leq g(n) \leq c\, g(n) \text{ for all } n \geq n_0\}.\end{aligned}$$

The notation gives an upper bound on a function, to within a constant factor. If $T(n) = \mathcal{O}(g(n))$ the value of $T(n)$ is on or below $g(n)$.

To illustrate the use of the $\mathcal{O}$-notation consider the modified *Arc-to-Arc-Floye-Warshall* algorithm shown in Algorithm 2.1. In the first part of the algorithm (lines 1–7), the predecessor and distance matrix is updated for each pair of arcs u and v where both are elements of A^*. The update is performed τ^2 times, where $\tau = |\boldsymbol{A}^*|$. In the second part of the algorithm (lines 8–13), $D(i,j) + D(k,j) + c(k)$ is compared to $D(i,j)$ for each $i, j, k \in \boldsymbol{A}^*$. The comparison is performed τ^3 times. The second part of the code dominates the first by an order of magnitude, so the algorithm has a worst-case running time of $\mathcal{O}(\tau^3)$. We use the $\mathcal{O}$-notation as a measure of the complexity of an algorithm, thus the *Arc-to-Arc-Floyd-Warshall* algorithm has a global complexity of $\mathcal{O}(\tau^3)$,[5]. If a different algorithm were to perform the same function in $\mathcal{O}(\tau^2)$ we could infer that the algorithm is more efficient than our *Arc-to-Arc-Floyd-Warshall* implementation.

2.C *Cen-IF* problem instance description

A Geospatial Information System (GIS) data set of the Centurion area, courtesy of *Business Connexion*, was used to create the benchmark instances. The data set accurately describes the network and includes a number of useful attributes. Accurate deadheading costs, service costs and waste quantities are not available for the network, so we inferred the arc-routing data using a similar approach to that of Belenguer et al. [6]. Though the metadata for the Centurion files are fabricated, the actual road network data are not. The municipality of Centurion has to service the entire road network, so the large files are representative of actual waste collection problems.

All road segment centerlines are represented by polylines, which, in turn, is made up out of no less than two nodes (or points). We infer the origin node, denoted by `FromNode`, as the first node in the polyline description, and the destination node, denoted by `ToNode`, as the last node. We refer to the `FromNode`–`ToNode` combination as a link.

[5] More accurately the *Arc-to-Arc-Floyd-Warshall* algorithm's complexity can be bounded by an asymptotically tight bound $\Theta(n^3)$. We do not give the Θ-notation's formal description as it is not applicable to other algorithms presented in this thesis. For a comprehensive discussion of algorithm growth functions, including Θ-notation, we refer the reader to [20].

Associated with each link is a ONEWAY field that has one of three values: 'B' indicates that it is a bi-directional road segment; 'FT' indicates a one-way in the direction from the FromNode to the ToNode; and 'TF' indicates a one-way in the direction from the ToNode to the FromNode. We infer the link to be an arc if it has a field value of either 'TF' or 'FT', and an edge if the field value is 'B'.

A road category field identifies the road type within the network hierarchy. If the link has a field value of type 'STREET' or 'OTHER', we assign a value 1 to the link, or a value 2 if the type is 'DUAL CARRIAGEWAY', 'NATIONAL HIGHWAY', 'NATIONAL ROAD', 'MAIN ROAD' or 'RESTRICTED ACCESS ROAD'. Both sides of a type 1 link are assumed to be serviceable in a single traversal. If a link is either an arc or edge with a type 1 value, it will remain a single link. We assume that the two sides of a type 2 link, on the other hand, must be serviced separately since it would be either dangerous, or physically impossible to have refuse collected on both sides of the road. An example would be a busy suburban road with two lanes in either direction. Links of type 2 that are arcs are then replaced with two arcs, both in the direction of the original arc, each arc representing one side of the road. Type 2 links that are edges are replaced with two directed arcs, one in either direction.

Links, both arcs and edges of either type 1 or 2, with a speed limit exceeding 60 km/h are assumed to have no demand. Links with a speed limit of at most 60 km/h are assumed to have demand. Demand of 10kg per household, and one household each 20 m is assumed. In the case of type 1 links the demand is doubled since both sides of the road is assumed to have demand. Demand is then calculated using Equation (2.11)

$$\text{demand} = \frac{d}{20} \times 10 \times m \tag{2.11}$$

where d denotes the length of the link, and m the multiplication factor of 2 if it is of type 1, and 1 if it is of type 2.

Road segments with a category other than those listed are assumed to be of type 1, but with no demand.

The traversal cost of a link is determined as the time it takes to traverse the link at a speed of 20 km/h. To derive the service cost, we add a loading time of 10 seconds per bin—10 kg or part thereof—to the traversal cost.

We impose a maximum vehicle capacity of 10 tons (10 000 kg) and a maximum trip length of 8 hours (28 800 seconds). For each street segment in the network we model separate deadheading and collection times. We also assume that a intermediate facility visit incurs a cost of 300 seconds.

Chapter 3

Splitting procedures

Since the Capacitated Arc Routing Problem (CARP) and all its extensions are $\mathcal{NP}$-hard the most effective methods for solving the problems are based on heuristics and metaheuristics, many of which employ giant tour approaches that rely on tour splitting procedures [70]. In this chapter we present optimal and heuristic splitting procedures for the Mixed Capacitated Arc Routing Problem under Time Restrictions with Intermediate Facilities (MCARPTIF). The structure of our optimal splitting procedure provides a substantial improvement in efficiency over the existing Capacitated Arc Routing Problem under Time Restrictions with Intermediate Facilities (CARPTIF) version that we adapted for mixed networks. Fast, near-optimal splitting procedures are also presented. The procedures developed in this chapter have been published in Willemse and Joubert [89], and are used in the giant tour based constructive heuristics of the next chapter.

3.1 Introduction

Splitting procedures take as input a giant tour and partition the tour into feasible vehicle routes. In this chapter we develop an efficient optimal and two heuristic splitting procedures for the MCARPTIF. The optimal and near-optimal procedures were tested in a multi-start *Route-First-Cluster-Second* heuristic on large MCARPTIF instances. Tight time-limits were imposed to reduce the number of starts of the slower, optimal procedures compared to the faster near-optimal procedures. Even with less starts, the *Route-First-Cluster-Second* heuristic linked with our efficient optimal splitting procedure performed marginally better than the near-optimal splitting versions. However, the optimal procedure is more difficult to implement, making the near-optimal procedure a worthy substitute, especially for practical applications where problem instances are much larger.

In the next section we review current splitting procedures for the CARP and a few of its extensions. In Section 3.3 we present detailed descriptions of our splitting procedures. In Section 3.4 we report on computational experiments, focusing on the execution times of the procedures and the difference in partition costs between optimal and near-optimal splitting. We then compare the performance of the different procedures within a multi-start *Route-First-Cluster-Second* heuristic.

3.2 Splitting procedures for the CARP and CARPTIF

The first optimal splitting procedure for the CARP was developed by Ulusoy [82] as part of a *Route-First-Cluster-Second* constructive heuristic. The heuristic is similar to the one of Beasley [5] for the Vehicle Routing Problem. First, edge demands are ignored and a giant tour is constructed servicing all the required edges in $\boldsymbol{G}$. In the second phase, an auxiliary Directed Acyclic Graph (DAG) is constructed whose arcs represent feasible sub-tours of the giant tour, with respect to demand of the sub-tour and vehicle capacity. The DAG is constructed in such a way that the shortest path through the graph gives the optimal partition of the giant tour into feasible vehicle routes. The shortest path can be calculated using any shortest path algorithm. The splitting procedure consists of constructing the DAG, calculating the shortest path through the graph, and decoding the shortest path to retrieve the optimal giant tour partitions. Lacomme et al. [51] and Belenguer et al. [6] develop multi-start *Route-First-Cluster-Second* heuristics for the CARP and Mixed Capacitated Arc Routing Problem (MCARP), respectively, whereby different giant tours are constructed and partitioned, and the best returned as the final solution.

Ghiani et al. [32] develop a splitting procedure, similar to CARP versions, for the Capacitated Arc Routing Problem with Intermediate Facilities (CARPIF). Their procedure calculates the optimal placement of Intermediate Facility (IF) visits within a route. The problem allows intra-route offloads so that collected demand between IF visits never exceeds vehicle capacity, but it does not impose route duration limits. As such, a solution always consists of only one route. When a route duration limit is imposed the problem generalises to the CARPTIF. To solve the problem Ghiani et al. [31] develop a splitting procedure that constructs two DAGs. The first consists of multiple source and destination vertices, each representing a start- and end-edge of a sub-tour in the giant tour. Shortest paths through the DAG between the sources and destinations represent the optimal placement of IFs in all possible sub-tours. The shortest path costs, calculated using a shortest path algorithm, are then used to construct a second DAG whose shortest path represents the optimal partition of the giant tour into vehicle routes. The optimal placement of IFs in each route is traced back to the shortest paths in the first DAG. To our knowledge, the CARPTIF on a mixed network has not been formally studied in literature. The splitting procedure of Ghiani et al. [31] can however be applied as-is to giant tours on mixed networks. A solution for the Mixed Capacitated Arc Routing Problem under Time Restrictions with Intermediate Facilities (MCARPTIF) can thus be obtained by combining the Route-First phase of Belenguer et al. [6] for the MCARP to construct a giant tour on a mixed network, and then applying the CLARPIF splitting procedure of Ghiani et al. [31] for the Cluster-Second phase.

To improve the efficiency of splitting procedures Lacomme et al. [51] develop a compact procedure for the CARP that does not explicitly construct the DAG. Instead, the shortest path through the DAG is directly calculated when scanning sub-tours for their feasibility with respect to vehicle capacity limits. Their version also accounts for a secondary objective of minimising fleet size. The compact version is exclusively used in Memetic Algorithms for the CARP [51, 54, 81] and MCARP [6], which are currently some of the most effective solution methods for the problems. Memetic Algorithms are metaheuristics based on genetic algorithms enhanced with local search procedures. Chromosomes are encoded as giant tours and an optimal splitting procedure is used to determine chromosome fitness each time a new chromosome is evaluated. An efficient splitting procedure is critical for the applications since chromosome evaluation occurs tens of thousands of times during the Memetic Algorithm's execution.

In this chapter we extended the compact splitting version of Lacomme et al. [51] to the CARPIF on a mixed network (MCARPIF). We then further extended this version to deal with the MCARPTIF and show that it provides a substantial improvement in efficiency over the version of Ghiani et al. [31]. We also developed two quick heuristic splitting procedures, one that greedily inserts IF visits into sub-tours and then calculates the route partitions and a second that employs a *Next-Fit* bin-packing procedure.

3.3 New splitting procedures

Consistent with our MCARPTIF notations, introduced in Section 2.2.4 and summarised in Tables 2.1 and 2.2, the splitting procedures take as input a giant tour $\boldsymbol{S}$ to be partitioned, which consists of a list of tasks, $[S_1, \ldots, S_{|\boldsymbol{S}|}]$. It is assumed that the shortest path is always followed between consecutive tasks, the duration of which is given by D, and only required arcs, given by the set $\boldsymbol{R}$, are included in $\boldsymbol{S}$, thus it contains no depot or IF dummy arcs as these are implicitly accounted for by the splitting procedures. Sub-tours in $\boldsymbol{S}$ are denoted as $\boldsymbol{S}_{i \to j} = [S_i, \ldots, S_j]$ where $1 \leq i < j \leq n$ and $n = |\boldsymbol{S}|$. A single MCARPIF or MCARPTIF route is a list of tasks that always starts with the dummy depot task, σ, ends with a dummy IF and depot task, and may include intra-route IF visits. The splitting procedures always consider the best IF to visit between consecutive tasks u and v, which is pre-calculated and given by $\Phi^*(u, v)$. For convenience, instead of using Equation (2.3) the duration of the best IF visit is given as $\mu^*_{\text{IF}}(u, v)$, which includes offloading time, λ, and is calculated as Equation (3.1):

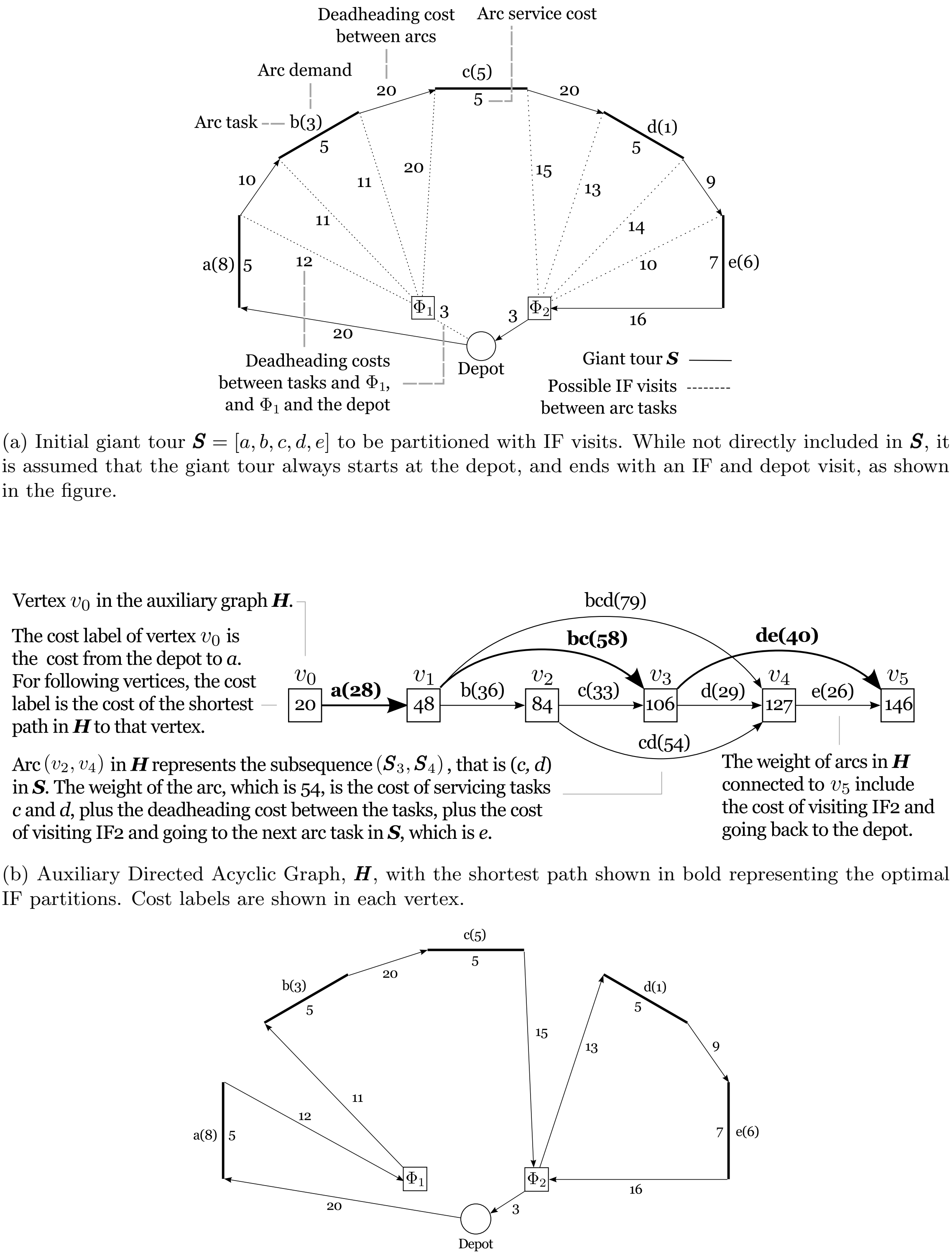
$$\mu^*_{\text{IF}}(u, v) = D\big(u, \Phi^*(u, v)\big) + D\big(\Phi^*(u, v), v\big) + \lambda. \tag{3.1}$$

The list of tasks between dummy depot and IF arcs represent subtrips and the load collected on a subtrip may not exceed Q. For the MCARPTIF the total duration of a route, including all task service times, deadheading travel time between tasks and the durations of IF visits, may not exceed a time restriction, L.

3.3.1 Splitting procedures for the MCARPIF

The first splitting procedure that we present is a Mixed Capacitated Arc Routing Problem with Intermediate Facilities (MCARPIF) adaptation of the CARP procedure developed by Lacomme et al. [51]. Recall that splitting procedures use $\boldsymbol{S}$ to construct an auxiliary DAG, $\boldsymbol{H}$, in such a way that its shortest path represents the optimal giant tour partition. For the CARPIF, Ghiani et al. [32] construct $\boldsymbol{H}$ by including a vertex for each feasible sub-tour $\boldsymbol{S}_{i \to j}$ with respect to vehicle capacity. Vertices representing consecutive sub-tours $\boldsymbol{S}_{i \to j}$ and $\boldsymbol{S}_{j+1 \to k}$ are then linked with arcs. A source vertex, linked to vertices representing $\boldsymbol{S}_{1 \to j}$, and a sink vertex, linked to vertices representing $\boldsymbol{S}_{k \to |\boldsymbol{S}|}$, are also included in $\boldsymbol{H}$ and the shortest path from the source to the sink represents the optimal partition. Using this approach the DAG consists of at most $\frac{n(n+1)}{2}$ vertices and $\frac{n(n+1)(n-1)}{6} + 2n$ arcs. Bellman's algorithm [20] can then compute the shortest path through $\boldsymbol{H}$ in $\mathcal{O}(n^3)$.

To develop a more efficient $\mathcal{O}(n^2)$ splitting procedure for the MCARPIF, the procedure of Lacomme et al. [51] can be adapted to construct a smaller sized DAG. Figure 3.1 shows the partitioning of $\boldsymbol{S} = [a, b, c, d, e]$ into a feasible MCARPIF route with sub-tours. The giant tour of Figure 3.1a is partitioned by building $\boldsymbol{H} = (\boldsymbol{V}', \boldsymbol{A}')$, with $n + 1$ vertices, with $\{v_0, \ldots, v_n\} \in \boldsymbol{V}'$, indexed from 0 onward (Figure 3.1b). Each sub-tour, $\boldsymbol{S}_{i \to j}$, with load less than Q represents a feasible sub-tour and is modelled as $(v_{i-1}, v_j) \in \boldsymbol{A}'$. Only feasible sub-tours are modelled in $\boldsymbol{A}'$. The cost of arc (v_{i-1}, v_j) is equal to the sub-tour cost, which includes the service cost of its tasks, the deadheading cost between tasks, and the cost of visiting an IF facility after S_j and travelling to S_{j+1} if $j < n$, or travelling from S_j to an IF and then the depot if $j = n$. For example, the feasible sub-tour $\boldsymbol{S}_{2 \to 4} = [b, c, d]$ is modelled by the arc (v_1, v_4) in $\boldsymbol{H}$, and its weight is 79, which is the cost of servicing tasks b, c and d, equalling $5 + 5 + 5 = 15$, plus the deadheading costs between tasks, equalling



(a) Initial giant tour $\boldsymbol{S} = [a, b, c, d, e]$ to be partitioned with IF visits. While not directly included in $\boldsymbol{S}$, it is assumed that the giant tour always starts at the depot, and ends with an IF and depot visit, as shown in the figure.

Vertex v_0 in the auxiliary graph $\boldsymbol{H}$.

The cost label of vertex v_0 is the cost from the depot to *a*. For following vertices, the cost label is the cost of the shortest path in $\boldsymbol{H}$ to that vertex.

Arc (v_2, v_4) in $\boldsymbol{H}$ represents the subsequence $(\boldsymbol{S}_3, \boldsymbol{S}_4)$, that is (*c, d*) in $\boldsymbol{S}$. The weight of the arc, which is 54, is the cost of servicing tasks *c* and *d*, plus the deadheading cost between the tasks, plus the cost of visiting IF2 and going to the next arc task in $\boldsymbol{S}$, which is *e*.

The weight of arcs in $\boldsymbol{H}$ connected to v_5 include the cost of visiting IF2 and going back to the depot.

(b) Auxiliary Directed Acyclic Graph, $\boldsymbol{H}$, with the shortest path shown in bold representing the optimal IF partitions. Cost labels are shown in each vertex.

(c) Optimally partitioned route with IF visits, obtained through $\boldsymbol{H}$. The total cost of the partitioned route is 146, which is equal to the shortest path from v_0 to v_5 in $\boldsymbol{H}$.

Figure 3.1: Example of a splitting procedure for the MCARPIF, with vehicle capacity $Q = 9$ and unloading cost $\lambda = 0$.

$20 + 20 = 40$, plus the cost of going to Φ_2 and to task e, equalling $14 + 10 = 24$. The shortest path from vertex 0 to n in $\boldsymbol{H}$ represents the optimal partition of $\boldsymbol{S}$ with IF visits (Figure 3.1c). Once $\boldsymbol{H}$ has been constructed, Bellman's algorithm for DAGs can efficiently compute the shortest path from vertex v_0 to v_n in $\boldsymbol{H}$.

The approach can be further improved by using a compact splitting procedure that does not explicitly construct $\boldsymbol{H}$, similar to the version of Lacomme et al. [51] for the CARP. Algorithm 3.1 shows such procedure, called *Efficient-IF-Split*, that we adapted for the MCARPIF. It directly calculates the optimal partition and further minimises the number of subtrips as a second objective. Modifications to the CARP procedure of Lacomme et al. [51] are contained in Lines 1, and 9–19, with the CARP calculations shown as comments (//) in Lines 2, 13, 16 and 19. Three labels are used by the procedure. The first, $\boldsymbol{N}_i$, represents the cost of the shortest path from vertex zero to i in $\boldsymbol{H}$, the second, $\boldsymbol{\Pi}_i$, represents the number of sub-tours in the same shortest path, and the third, $\boldsymbol{P}_i$, represents the predecessor vertex of i on this path and thus stores the resulting optimal placement of IFs in $\boldsymbol{S}$. Note that both $\boldsymbol{N}$ and $\boldsymbol{P}$ are indexed from zero. The depot and final IF and depot visit on a feasible route are implicitly accounted for by the algorithm. By increasing i and j the procedure successively scans sub-tours for capacity violations. When the feasible sub-tour $\boldsymbol{S}_{i \to j}$ is found the optimal partition for the partial giant tour

Algorithm 3.1: *Efficient-IF-Split* for the MCARPIF

```
   Input  : $\boldsymbol{S}$
   Output: $\boldsymbol{N}, \boldsymbol{P}$
 1 $n = |\boldsymbol{S}|$; $\boldsymbol{\Pi}_0 = 0$; $\boldsymbol{P}_0 = 0$; $\boldsymbol{N}_0 = D(\sigma, S_1)$;
 2 //* $\boldsymbol{N}_0 = 0$;
 3 for $i \leftarrow 1$ to $n$ do $\boldsymbol{N}_i = \infty$; $\boldsymbol{\Pi}_i = \infty$;
 4 for $i \leftarrow 1$ to $n$ do
 5     $l' = 0$; $c' = 0$; $j = i$;
 6     repeat
 7         $l' = l' + q(S_j)$;
 8         if $l' \leq Q$ then
 9             if $j = n$ then
10                 $a = \mu^*_{\mathrm{IF}}(S_j, \sigma)$
11             else
12                 $a = \mu^*_{\mathrm{IF}}(S_j, S_{j+1})$
13             //* $a = D(S_j, \sigma)$;
14             if $i = j$ then
15                 $c' = w(S_j) + a$;
16                 //* $c' = D(\sigma, S_j) + w(S_j) + a$;
17             else
18                 $\Delta c' = D(S_{j-1}, S_j) - \mu^*_{\mathrm{IF}}(S_{j-1}, S_j)$;
19                 //* $\Delta c' = D(S_{j-1}, S_j) - D(S_{j-1}, \sigma)$;
20                 $c' = c' + \Delta c' + w(S_j) + a$;
21             if $(\boldsymbol{N}_{i-1} + c' < \boldsymbol{N}_j)$ or $\big((\boldsymbol{N}_{i-1} + c' = \boldsymbol{N}_j)$ and $(\boldsymbol{\Pi}_{i-1} + 1 < \boldsymbol{\Pi}_j)\big)$ then
22                 $\boldsymbol{N}_j = \boldsymbol{N}_{i-1} + c'$;
23                 $\boldsymbol{\Pi}_j = \boldsymbol{\Pi}_{i-1} + 1$;
24                 $\boldsymbol{P}_j = i - 1$;
25             $j = j + 1$;
26     until $(j > n)$ or $(l' > Q)$;
27 return $(\boldsymbol{N}, \boldsymbol{P})$
```

*Original calculations of Lacomme et al. [51] for the CARP.

ending at j is updated. In the worst case a total of $\frac{n(n+1)}{2}$ sub-tours can be evaluated, giving the algorithm a running time of $\mathcal{O}(n^2)$. The actual running time of the algorithm is reduced as only feasible sub-tours that meet the capacity constraint are evaluated.

3.3.2 Splitting procedures for the MCARPTIF

For the MCARPTIF the splitting procedure has to simultaneously determine the optimal IF partitions, resulting from the vehicle capacity limit Q, and the optimal route partitions, resulting from the route duration restriction, L. Ghiani et al. [31] extend the method of Ghiani et al. [32] and explicitly construct two DAGs for this purpose. *Efficient-IF-Split* can be used in the same way. First, it calculates the optimal IF partitions for *all* the sub-tours in $\boldsymbol{S}$. The cost of the partitioned sub-tours are then used to construct the second DAG, and the shortest path through it represents the optimal route partitions. We refer to this splitting version for the MCARPTIF as *Two-Phase-Split*.

Two-Phase-Split

To illustrate the *Two-Phase-Split* procedure, the same giant tour in Figure 3.1a is used and a route duration restriction of $L = 70$ is imposed. It is further assumed that the shortest paths from the depot to tasks in $\boldsymbol{S}$, excluding task a, are through the respective IFs. For the first DAG, $\boldsymbol{H}$, each of the $\frac{n(n-1)}{2}$ sub-tours of $\boldsymbol{S}$ is included in $\boldsymbol{H}$, and each one is treated as a single route, starting at the depot, and ending with an IF and depot visit. Each sub-tour can be optimally partitioned with IF visits using *Efficient-IF-Split*. The result of the first phase on the example route is shown in Figure 3.2, in which the optimal partition of each sub-tour is calculated and given separately.

Next, as shown in Figure 3.3, the algorithm uses the feasible sub-tours with durations exceeding L to construct the second DAG, $\boldsymbol{H'} = (\boldsymbol{V''}, \boldsymbol{A''})$, with $n+1$ vertices, $\{v_0, \ldots, v_n\} \in \boldsymbol{V''}$, indexed from zero onward (Figure 3.3a). Each sub-tour $\boldsymbol{S}_{i\rightarrow j}$ is modelled by one arc (v'_{i-1}, v'_j) in $\boldsymbol{A''}$, weighted by the route's partitioned cost, given by the cost of the shortest path in $\boldsymbol{H}$ from v_{i-1} to v_j. In Figure 3.3a, arc (v'_0, v'_2) in $\boldsymbol{H'}$ models the route starting with task $S_1 = a$ and ending with task $S_2 = b$. Referring back to $\boldsymbol{H}$ (Figure 3.2), the optimal IF partition cost of this route is 67, which becomes the weight of arc (v'_0, v'_2) in $\boldsymbol{H'}$. Since the IF partitioned cost of the route is less than L, it is included in $\boldsymbol{H'}$. The shortest path from vertex v'_0 to v'_n in $\boldsymbol{H'}$ represents the optimal route partition of $\boldsymbol{S}$, and the optimal IF visits for each route can be traced back to $\boldsymbol{H}$. The optimal route and IF partitions for the giant tour in Figure 3.1a are shown in Figure 3.3b. When constructing $\boldsymbol{H}$, the optimal placement of IFs does not have to be calculated for all sub-tours. If the optimally partitioned sub-tour $\boldsymbol{S}_{i\rightarrow j}$ exceeds L, then all the partitioned sub-tours $\boldsymbol{S}_{i\rightarrow j+k}$, where $0 < k \leq n-j$, will also exceed L and can thus be skipped in the construction of $\boldsymbol{H}$.

Efficient-Split

Similar to *Efficient-IF-Split* for the MCARPIF, $\boldsymbol{H}$ and $\boldsymbol{H'}$ need not be explicitly constructed. Algorithm 3.2 shows a compact splitting version for the MCARPTIF that directly calculates the optimal partitions for subtrips and routes. We refer to this version as *Efficient-Split*. Algorithm 3.2 minimises the total number of routes as the primary objective, and partition cost as secondary. For the IF partitions, two labels are used for each vertex i in $\boldsymbol{H}$. The first, $\boldsymbol{N}_{i,j}$, where $i \leq j$, represents the cost of the shortest path from vertex i to j in $\boldsymbol{H}$, and the second, $\boldsymbol{P}_{i,j}$, where $i \leq j$, represents the predecessor vertex of j

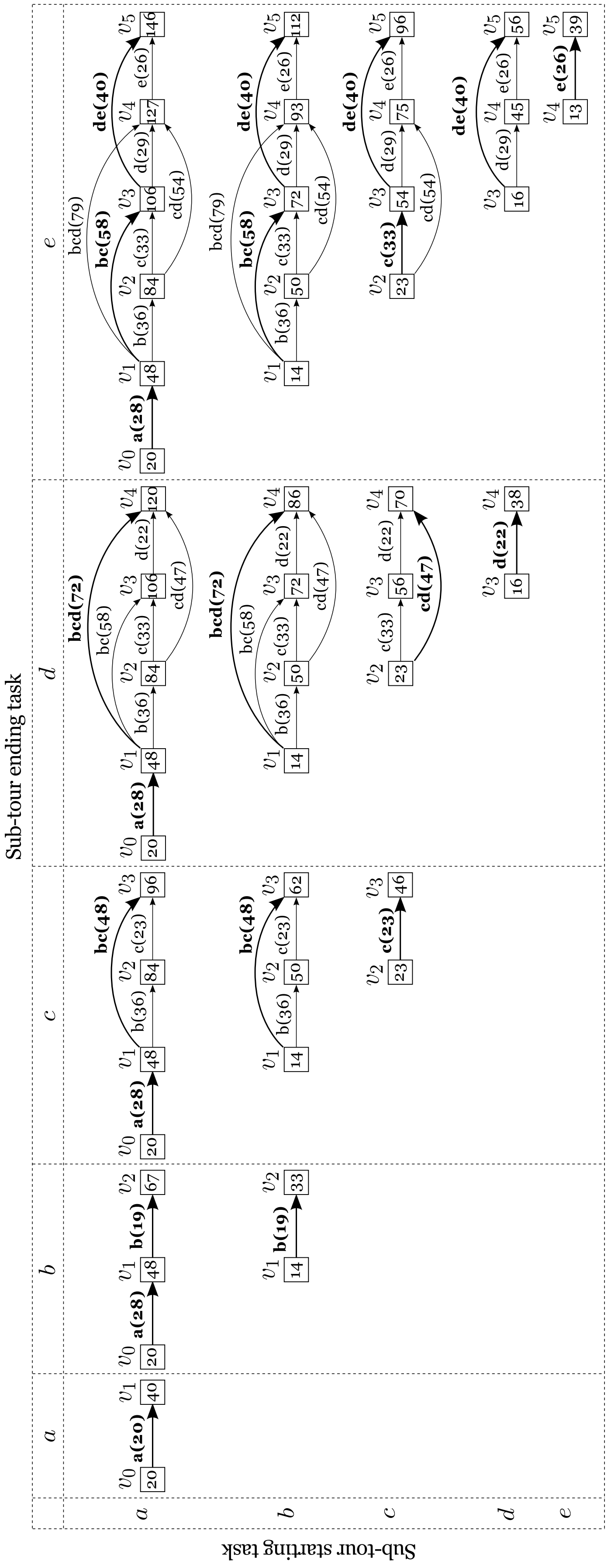
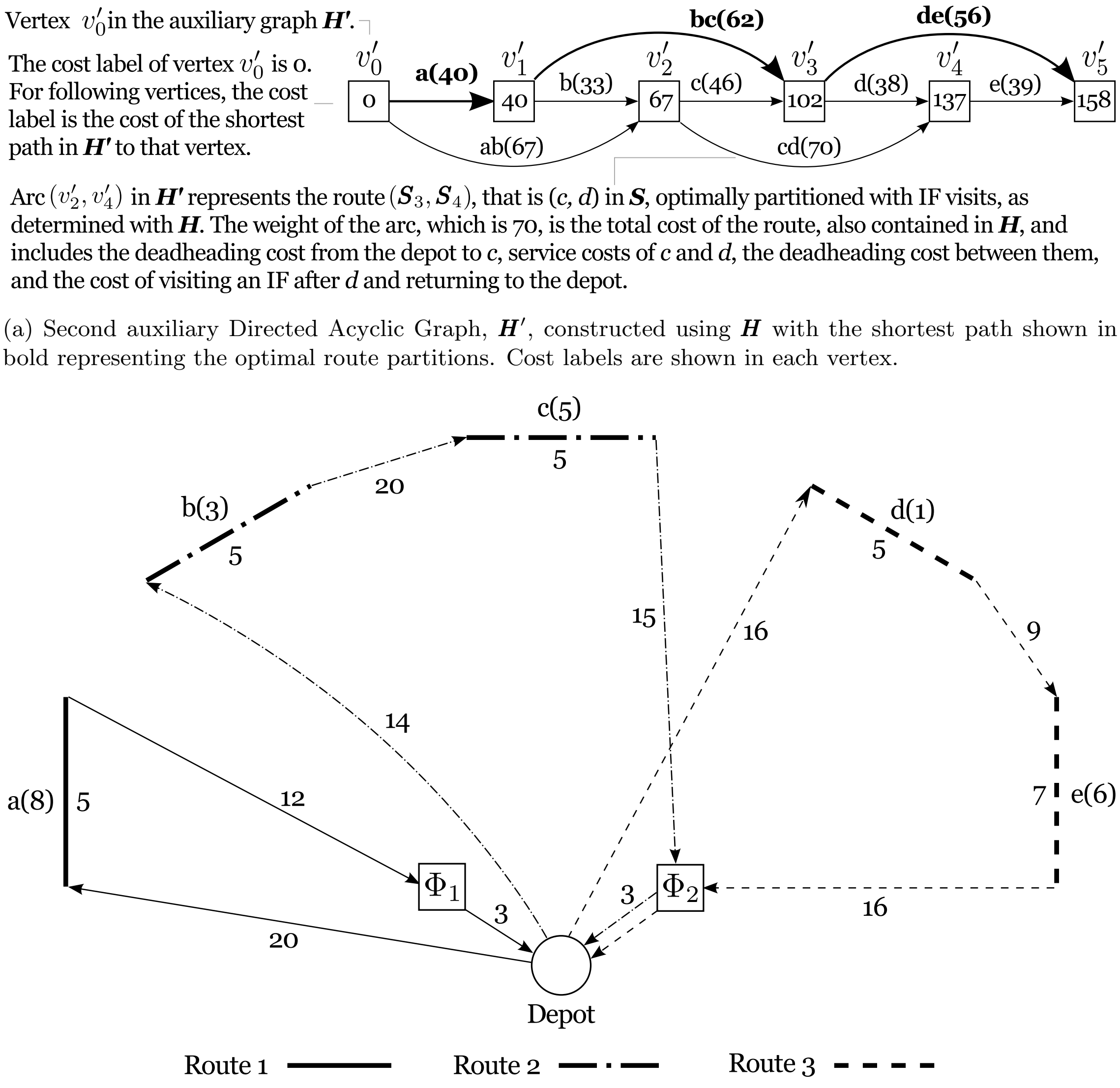

Sub-tour ending task
Sub-tour starting task
a
b
c
d
e
v_0 20 a(20) v_1 40
v_0 20 a(28) v_1 48 b(19) v_2 67
v_1 14 b(19) v_2 33
v_0 20 a(28) v_1 48 b(36) v_2 84 c(23) v_3 96 bc(48)
v_1 14 b(36) v_2 50 c(23) v_3 62 bc(48)
v_2 23 c(23) v_3 46
v_0 20 a(28) v_1 48 b(36) v_2 84 c(33) v_3 106 d(22) v_4 120 bc(58) bcd(72) cd(47)
v_1 14 b(36) v_2 50 c(33) v_3 72 d(22) v_4 86 bc(58) bcd(72) cd(47)
v_2 23 c(33) v_3 56 d(22) v_4 70 cd(47)
v_3 16 d(22) v_4 38
v_0 20 a(28) v_1 48 b(36) v_2 84 c(33) v_3 106 d(29) v_4 127 e(26) v_5 146 bc(58) bcd(79) cd(54) de(40)
v_1 14 b(36) v_2 50 c(33) v_3 72 d(29) v_4 93 e(26) v_5 112 bc(58) bcd(79) cd(54) de(40)
v_2 23 c(33) v_3 54 d(29) v_4 75 e(26) v_5 96 cd(54) de(40)
v_3 16 d(29) v_4 45 e(26) v_5 56 de(40)
v_4 13 e(26) v_5 39

Figure 3.2: Optimal IF partitions, shown in bold, for all possible sub-tours in $\boldsymbol{S}$ of the example route.



Vertex v'_0 in the auxiliary graph $\boldsymbol{H'}$.

The cost label of vertex v'_0 is 0. For following vertices, the cost label is the cost of the shortest path in $\boldsymbol{H'}$ to that vertex.

Arc (v'_2, v'_4) in $\boldsymbol{H'}$ represents the route $(\boldsymbol{S}_3, \boldsymbol{S}_4)$, that is (*c, d*) in $\boldsymbol{S}$, optimally partitioned with IF visits, as determined with $\boldsymbol{H}$. The weight of the arc, which is 70, is the total cost of the route, also contained in $\boldsymbol{H}$, and includes the deadheading cost from the depot to *c*, service costs of *c* and *d*, the deadheading cost between them, and the cost of visiting an IF after *d* and returning to the depot.

(a) Second auxiliary Directed Acyclic Graph, $\boldsymbol{H}'$, constructed using $\boldsymbol{H}$ with the shortest path shown in bold representing the optimal route partitions. Cost labels are shown in each vertex.

(b) Optimally partitioned giant tour with each route further partitioned with IF visits, as given by the shortest paths in $\boldsymbol{H}$ and $\boldsymbol{H}'$

.

Figure 3.3: Example of a splitting procedure for the MCARPTIF, with vehicle capacity $Q = 9$, unloading time of $\lambda = 0$ and route duration restriction of $L = 70$.

Algorithm 3.2: *Efficient-Split* for the MCARPTIF

```
Input  : $\boldsymbol{S}$
Output: $\boldsymbol{N}, \boldsymbol{P}, \boldsymbol{N}', \boldsymbol{P}$'
$n = |\boldsymbol{S}|$;
for $i \leftarrow 0$ to $n-1$ do $\boldsymbol{N}_{i,i} = \boldsymbol{D}(\sigma, \boldsymbol{S}_i)$; $\boldsymbol{P}_{i,i} = i$ ;
for $i \leftarrow 1$ to $n-1$ do
    for $j \leftarrow i+1$ to $n$ do
        $\boldsymbol{N}_{i,j} = \infty$;
$\boldsymbol{\Pi}'_0 = 0$; $\boldsymbol{P}'_0 = 0$; $\boldsymbol{N}'_0 = 0$;
for $i \leftarrow 1$ to $n$ do $\boldsymbol{N}'_i = \infty$; $\boldsymbol{\Pi}_i = \infty$ ;
$k_s = 0$;
for $i \leftarrow 1$ to $n$ do
    $l' = 0$; $c' = 0$; $c'' = 0$; $j = i$; $k_0 = k_s$;
    repeat
        $l' = l' + q(S_j)$;
        if $l' \leq Q$ then
            if $j = n$ then $a = \mu^*_{\mathrm{IF}}(S_j, \sigma)$ ;
            else $a = \mu^*_{\mathrm{IF}}(S_j, S_{j+1})$ ;
            if $i = j$ then
                $c' = w(S_j) + a$;
            else
                $\Delta c' = \boldsymbol{D}(S_{j-1}, S_j) - \mu^*_{\mathrm{IF}}(S_{j-1}, S_j)$;
                $c' = c' + \Delta c' + w(S_i) + a$;
            $c'' = c' + \mu^*_{\mathrm{IF}}(S_j, \sigma) - a$;
            if $c'' \leq L$ then
                for $k \leftarrow k_0$ to $i-1$ do
                    $N_{temp} = \boldsymbol{N}_{k,i-1} + c'$;
                    if $N_{temp} < \boldsymbol{N}_{k,j}$ then
                        $\boldsymbol{N}_{k,j} = N_{temp}$;
                        $\boldsymbol{P}_{k,j} = k$;
                    $N'_{temp} = \boldsymbol{N}_{k,i-1} + c''$;
                    if $N'_{temp} \leq L$ then
                        if $(\boldsymbol{\Pi}_i + 1 < \boldsymbol{\Pi}_j)$ or $\big((\boldsymbol{\Pi}_i + 1 = \boldsymbol{\Pi}_j)$ and $(\boldsymbol{N}'_k + N'_{temp} < \boldsymbol{N}'_j)\big)$ then
                            $\boldsymbol{\Pi}_j = \boldsymbol{\Pi}_i + 1$;
                            $\boldsymbol{N}'_j = \boldsymbol{N}'_k + N'_{temp}$;
                            $\boldsymbol{P}'_j = k$;
                    if $(i = j)$ and $(j < n)$ then
                        $N'_{best} = \boldsymbol{N}_{k,j} - \mu^*_{\mathrm{IF}}(S_j, S_{j+1}) + D(S_j, S_{j+1}) + w(S_{j+1}) + \mu^*_{\mathrm{IF}}(S_{j+1}, \sigma)$;
                        if $N'_{best} > L$ then $k_s = k_s + 1$ ;
                $j = j + 1$;
    until $(j > n)$ or $(l' > Q)$ or $(c'' > L)$;
return $(\boldsymbol{N}, \boldsymbol{P}, \boldsymbol{N}', \boldsymbol{P}$')
```

on this path back to i and thus stores the resulting optimal placement of IFs in sub-tours of $\boldsymbol{S}$. Two more labels are used for the optimal route partitions. The first, $\boldsymbol{N}'_i$, represents the cost of the shortest path from vertex zero to i in $\boldsymbol{H}'$, and the second, $\boldsymbol{P}'_i$, represents the predecessor vertex of i on this path and thus stores the resulting optimal route partitions of $\boldsymbol{S}$. Since the MCARPTIF has multiple routes, the partitioning algorithm is further extended by including $\boldsymbol{\Pi}_i$, which represents the number of routes required from vertex zero to i in $\boldsymbol{H}'$. By increasing i and j the procedure successively scans sub-tours for capacity violations. When the feasible sub-tour $\boldsymbol{S}_{i\to j}$ is found the optimal partitions for all sub-tours starting at $k \in \{1, \ldots, i\}$ and ending at j are updated. In the worst case, a total of $\frac{n(n+1)(n+2)}{6}$ sub-tours are evaluated for the optimal IF partitions, giving the algorithm a running time of $\mathcal{O}(n^3)$. The actual running time of Algorithm 3.2 is reduced as only feasible sub-tours, adhering to the route duration restriction, are optimally partitioned with IF visits. When $i = j$, the optimal partitions of all sub-tours from $k \in \{1, \ldots, j\}$ to j have been calculated. If $\boldsymbol{S}_{k\to j}$ then exceeds L, longer sub-tours starting at k will also exceed L and they need not be updated with IF partitions in subsequent iterations. This significantly reduces the number of sub-tours that have to be updated each time i and j are incremented.

3.3.3 Heuristic splitting procedures for the MCARPTIF

The last two splitting procedures that we developed for the MCARPTIF are heuristic in nature and thus do not guarantee an optimal partition. The first is a straight forward *Next-Fit* bin-packing type procedure that starts with a route consisting of the starting depot task. Starting with the first task in $\boldsymbol{S}$, tasks are progressively added in sequence to the route. If a task cannot be added without exceeding Q, an IF visit is included before the task. If a task cannot be added to the route without exceeding L, including the time of going from the task to the nearest IF and depot, the route is closed and the task is added to a new route. The procedure, which we call *Simple-Split*, runs in $\mathcal{O}(n)$.

The second heuristic procedure that we developed, called *Heuristic-Split*, is shown in Algorithm 3.3. It is an extension of *Efficient-IF-Split* whereby IF visits are included in routes greedily instead of optimally. When evaluating progressively longer sub-tours, IF visits are inserted the moment that the load of a sub-tour since the last IF insertion exceeds the vehicle capacity. The optimal placement of IFs is not calculated for each sub-tour, reducing the computational complexity of the algorithm to $\mathcal{O}(n^2)$ while still ensuring that it produces feasible routes with respect to Q and L. After partitioning $\boldsymbol{S}$, the IF partitions for each of the resulting routes can be improved using *Efficient-IF-Split* while maintaining the overall complexity of $\mathcal{O}(n^2)$.

3.4 Computational results

For the computational tests we analysed and compared the efficiency of *Two-Phase-Split*, *Efficient-Split*, *Simple-Split* and *Heuristic-Split* for the MCARPTIF. Efficiency was measured as the time taken by each procedure to partition a giant tour. We then compared the number of routes and cost of the giant tour partitions from each procedure. *Heuristic-Split* was always linked with *Efficient-IF-Split* as a post-partition procedure to improve the IF placements in each route. The aim was to determine if *Simple-Split* or *Heuristic-Split* could be potentially used as a substitute for *Two-Phase-Split* and *Efficient-Split* in giant tour solution methods when solving large problem instances.

Two sets of MCARPTIF benchmark problems, introduced in Section 2.4.2, were used

Algorithm 3.3: *Heuristic-Split* for the MCARPTIF

```
Input  : $\boldsymbol{S}$
Output: $\boldsymbol{N}, \boldsymbol{P}$
$n = |\boldsymbol{S}|$; $\boldsymbol{P}_0 = 0$; $\boldsymbol{N}_0 = 0$;
$\boldsymbol{\Pi}'_0 = 0$; for $i \leftarrow 1$ to $n$ do  $\boldsymbol{N}'_i = \infty$; $\boldsymbol{\Pi}_i = \infty$ ;
for $i \leftarrow 1$ to $n$ do
    $l' = 0$; $c' = 0$; $j = i$;
    repeat
        $l' = l' + q(S_j)$;
        if $i = j$ then
            $c' = D(\sigma, S_j) + w(S_j) + \mu^*_{\mathrm{IF}}(S_j, \sigma)$;
        else
            if $l' \leq Q$ then
                $\Delta c' = D(S_{j-1}, S_j) - \mu^*_{\mathrm{IF}}(S_{j-1}, \sigma)$
            else
                $\Delta c' = \mu^*(S_{j-1}, S_j) - \mu^*_{\mathrm{IF}}(S_{j-1}, \sigma)$;
                $l' = q(S_j)$;
            $c' = c' + \Delta c' + w(S_j) + \mu^*_{\mathrm{IF}}(S_j, \sigma)$;
        if $c' \leq L$ then
            if $(\boldsymbol{\Pi}_i + 1 < \boldsymbol{\Pi}_j)$ or $\big((\boldsymbol{\Pi}_i + 1 = \boldsymbol{\Pi}_j)$ and $(\boldsymbol{N}_{i-1} + c' < \boldsymbol{N}_j)\big)$ then
                $\boldsymbol{\Pi}_j = \boldsymbol{\Pi}_i + 1$;
                $\boldsymbol{N}_j = \boldsymbol{N}_{i-1} + c'$;
                $\boldsymbol{P}_j = i - 1$;
            $j = j + 1$;
    until $(j > n)$ or $(c' > L)$;
return $(\boldsymbol{N}, \boldsymbol{P})$
```

for our tests. The first set, *Lpr-IF*, is based on the *lpr* MCARP set of Belenguer et al. [6]. The set contains random networks that were generated in such a way to mimic waste collection routing. The second data set used was the *Cen-IF* set that contains three very large benchmark instances, based on actual road networks of Centurion. All heuristics were programmed in Python version 2.7, with critical procedures optimised using Cython version 0.17.1. Computational experiments were run on a Macbook Pro with a 2.5 Ghz Intel Core i5 processor and with 8 GB memory.

To compare the efficiency of the procedures thirty giant tours were constructed using the *Path-Scanning-Random-Link* constructive heuristic, developed by Belenguer et al. [6] for the MCARP, and relaxing the capacity limit and route length restriction. The constructive heuristic is described in more detail in the next chapter in Section 4.3.1. The giant tours were then partitioned using one of the four splitting procedures with the primary objective to minimise the number of required routes, and secondly to minimise the partition cost. Figure 3.4 shows the CPU time taken by each procedure to partition the giant tours as a function of the size of the tours, n. Giant tours for *Cen-IF* problems are of

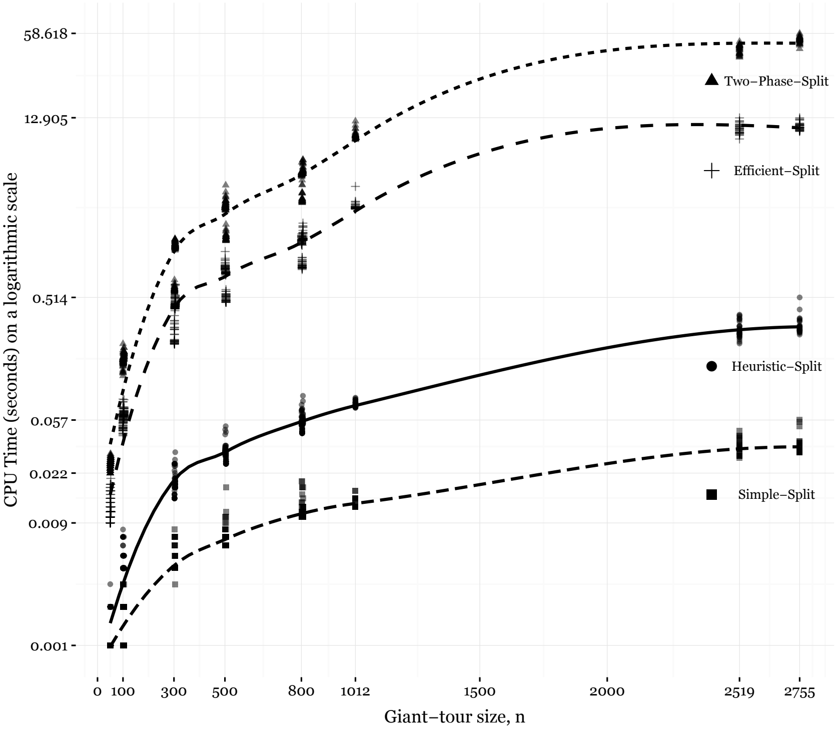


Figure 3.4: Scatter plots and trend lines for giant tour size, n, versus partitioning time (in seconds, on a logarithmic scale) of four MCARPTIF splitting procedures on thirty different giant tours per instance.

sizes 1012, 2519 and 2755. All other points in the figure are for the *Lpr-IF* instances. All four splitting procedures show polynomial growth in their execution times as a factor of n. *Simple-Split* was extremely efficient, since it runs in $\mathcal{O}(n)$, splitting the *Cen-IF* tours with over 2500 tasks in less than 0.05 seconds. *Heuristic-Split* was also efficient, taking less than 0.5 seconds to partition the *Cen-IF* tours. Owing to their $\mathcal{O}(n^3)$ complexity, *Two-Phase-Split* and *Efficient-Split* took longer to partition tours. On all instances, *Efficient-Split* was substantially quicker than *Two-Phase-Split*, though on small instances with $n \leq 104$ the difference in performance may not be practically significant, with both procedures taking less than 0.5 seconds per partition. *Two-Phase-Split* took close to 60 seconds on the largest *Cen-IF* problem, whereas *Efficient-Split* took only 12 seconds.

Next we compared the solution quality of the partitioning procedures over the thirty *Path-Scanning-Random-Link* giant tours per problem instance. Since *Two-Phase-Split* and *Efficient-Split* produce the same optimal partitions their partition costs and resulting number of routes were used as a target. For the primary objective, to minimise the number of routes resulting from the partition, both *Simple-Split* and *Heuristic-Split* matched *Two-Phase-Split* on more than 79 of the 90 *Cen-IF* and on 431 of the 450 *Lpr-IF* giant tours. On the remaining tours, the near-optimal procedures required at most one additional route. To see how close the suboptimal partitions were to the optimal partitions, cost wise, the cost gap between the partitions was measured as $Z_{\text{gap}} = \frac{Z - Z^*}{Z^*}$ where Z is the cost of the final partition of the giant tour, partitioned using *Simple-Split* or *Heuristic-Split*, and Z^* is the cost of the optimal partition, as calculated through *Efficient-Split*. Results per problem are shown in Figure 3.5 where each problem is labelled according to its giant tour size, n. Except for two problems, the partitions produced by *Heuristic-Split* were very close to optimal with an average cost gap of less than 1%. The cost gap range (maximum gap minus minimum gap) of the procedure is also small, being less than or close to 2%, which shows that it consistently produced near-optimal partitions. Negative cost gaps were observed for the procedure on some of the giant tours for *Cen-IF-c*. These were tours where *Heuristic-Split* partitions required an extra route over the optimal partitions, and the extra route resulted in a lower solution cost. Since minimising the number of routes was our primary objective, these partitions are suboptimal. *Simple-Split* matched

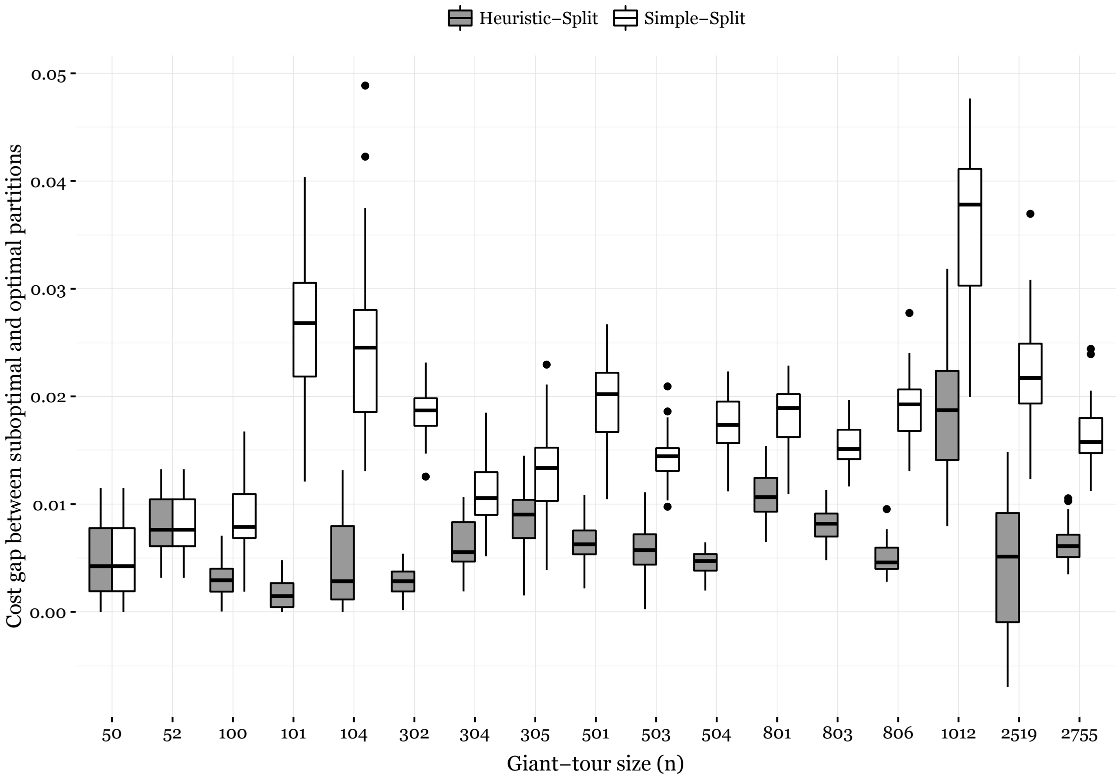


Figure 3.5: Box-and-Whisker plot for the cost gap, Z_{gap}, between the partitions of *Simple-Split* and *Heuristic-Split*, and the optimal partitions of *Efficient-Split*.

Heuristic-Split on the two smallest giant tour problems. On all other problems its cost gap was larger, which was expected given its simple structure. The cost gap range of *Simple-Split* was also larger than *Heuristic-Split*, showing that its performance was not as consistent.

Our results show that as a factor of giant tour size, the execution time of *Heuristic-Split* is an order of magnitude lower (faster) to that of *Efficient-Split*, whose computation time is an order of magnitude lower than *Two-Phase-Split* (Figure 3.4). Furthermore, *Heuristic-Split* consistently produced near-optimal partitions, whereas *Simple-Split* resulted in higher cost partitions and its performance varied per problem (Figure 3.5). In time-limited applications, methods linked with *Simple-Split* and *Heuristic-Split* may produce better results than *Two-Phase-Split* and *Efficient-Split* since their higher efficiency will allow the methods to evaluate and split more giant tours.

To test the hypothesis the following experiment was conducted using a multi-start *Route-First-Cluster-Second* constructive heuristic. *Path-Scanning-Random-Link* (PSRL) was again used for the Route-First phase and different splitting procedures were used for the Cluster-Second phase. Giant tours, representing different starts, were generated with PSRL and partitioned by the splitting procedure until an execution time-limit of 60 seconds was reached. The best partitioned solution from all the starts was then returned as the final solution. This was repeated for 30 experiments per problem and the mean number of routes, $\overline{K}$, and cost, $\overline{Z}$, over the returned solutions were calculated. Summary results on the *Lpr-IF* and *Cen-IF* problem sets are shown in Table 3.1. Within the

Table 3.1: Mean total cost results over 30 experiments for different splitting procedures used in a multi-start *Route-First-Cluster-Second* constructive heuristic. For each experiment the heuristic was executed until a 60 second time-limit was reached.

	Efficient-Split	*Two-Phase-Split*		*Heuristic-Split*		*Simple-Split*	
Instance	$\overline{Z}^*$	$\overline{Z}$	$\overline{Z}/\overline{Z}^*$	$\overline{Z}$	$\overline{Z}/\overline{Z}^*$	$\overline{Z}$	$\overline{Z}/\overline{Z}^*$
Cen-IF-a	240696	241511	1.003	241100	1.002	244564	1.016
Cen-IF-b	602695	612302	1.016	603543	1.001	606484	1.006
Cen-IF-c	533331	544645	1.021	532083	– (1)	538188	1.009
Mean			*1.014*		*1.002*		*1.010*
Lpr-IF-a-01	13594	13594	1.000	13684	1.007	13684	1.007
Lpr-IF-a-02	28392	28480	1.003	28569	1.006	28569	1.006
Lpr-IF-a-03	78835	78988	1.002	78842	1.000	79409	1.007
Lpr-IF-a-04	133130	133312	1.001	133630	1.004	134652	1.011
Lpr-IF-a-05	211765	212093	1.002	212337	1.003	214510	1.013
Lpr-IF-b-01	14835	14835	1.000	14871	1.002	14871	1.002
Lpr-IF-b-02	28897	28928	1.001	28926	1.001	29412	1.018
Lpr-IF-b-03	80222	80290	1.001	80419	1.002	80678	1.006
Lpr-IF-b-04	132254	132485	1.002	132366	1.001	134318	1.016
Lpr-IF-b-05	220638	221079	1.002	221175	1.002	223750	1.014
Lpr-IF-c-01	18734	18734	1.000	18765	1.002	18765	1.002
Lpr-IF-c-02	36596	36618	1.001	36596	1.000	36596	1.000
Lpr-IF-c-03	114012	114097	1.001	114093	1.001	115535	1.013
Lpr-IF-c-04	173817	174087	1.002	174214	1.002	176197	1.014
Lpr-IF-c-05	273427	273675	1.001	274325	1.003	276953	1.013
Mean			*1.001*		*1.002*		*1.013*

$\overline{Z}^*$: Mean solution cost for *Efficient-Split*; $\overline{Z}$: Mean solution cost for *Two-Phase-Split*, *Heuristic-Split* and *Simple-Split*. (1): *Heuristic-split* failed to produce the minimum fleet size on the *Cen-IF-c* instances.

multi-start heuristic all the procedures performed the same in minimising the number of solution routes. The only difference in performance was in minimising solution cost. As expected, *Efficient-Split* performed better than *Two-Phase-Split* since it was able to make more starts in 60 seconds while still producing optimal partitions. On small instances the difference between *Efficient-Split* and *Two-Phase-Split* was less prominent. *Heuristic-Split* outperformed *Two-Phase-Split* on all the *Cen-IF* instances, and five of the fifteen *Lpr-IF* problems. *Simple-Split* had up to 30 and 800 times more starts than *Efficient-Split* and *Heuristic-Split*, respectively, on large *Cen-IF* and *Lpr-IF* problems, but could only match *Efficient-Split* on one and *Heuristic-Split* on three of the smallest *Lpr-IF* problems. Solution costs of *Heuristic-Split* are within one percent of *Efficient-Split* and it performed better on the *Cen-IF-c* problem. On all other problems *Efficient-Split* performed the best, making it the best constructive heuristic implementation on both small and large problems, with and without execution time-limits imposed. The cost difference between the *Efficient-Split* and *Heuristic-Split* were small. Furthermore, as shown in Algorithm 3.2, implementing *Efficient-Split* is not trivial. The multi-start constructive heuristic linked with *Heuristic-Split* produced solutions close to *Efficient-Split*, and its structure is less complex (Algorithm 3.3) making it a worthy substitute to *Efficient-Split*, especially for practical applications.

3.5 Conclusion

As a first step to solve the MCARPTIF, efficient optimal and near-optimal splitting procedures were developed and tested on waste collection benchmark instances. Tests showed that *Efficient-Split* was significantly faster than *Two-Phase-Split.* On small instances the difference in performance between the two procedures was less prominent, highlighting the need for tests to be performed on more realistically sized problems. Two near-optimal splitting procedures were developed and tested, and results showed that *Heuristic-Split* was quicker than *Efficient-Split* while producing partitions that are close to optimal in terms of the number of required routes and total partition cost. Splitting procedures are typically called multiple times, either in constructing an initial solution for the MCARPTIF, or when embedded in Memetic Algorithms. Preliminary results for the former case showed that *Efficient-Split* performed the best in a multi-start constructive heuristic application. However, implementing *Efficient-Split* is not trivial, and given the simpler structure and performance of *Heuristic-Split*, we recommend that it be used as a substitute.

In the next chapter *Heuristic-Split* and *Efficient-Split* are further evaluated within a multi-start constructive heuristic application. We also adapt and compare other constructive heuristics for the MCARPTIF. Solutions from the constructive heuristics are then used as input for the improvement heuristics and metaheuristic developed in Chapters 5 to 7.

Chapter 4

Constructive heuristics

In this chapter we develop and test constructive heuristics for the Mixed Capacitated Arc Routing Problem under Time Restrictions with Intermediate Facilities (MCARPTIF). Our aim was to identify the best performing heuristics, which can then be used in waste collection routing applications when short execution times are required. The developed constructive heuristics have been published in [87], and are used in Chapters 5 to 7 to compute initial solutions for Local Search improvement heuristics and a Tabu Search metaheuristic.

4.1 Introduction

Constructive heuristics represent an important component of solution methods for Capacitated Arc Routing Problems (CARPs). In cases where execution time is limited, the heuristics generally provide good solutions in acceptable CPU time. The heuristics also provide initial solutions to be improved using metaheuristics. In this chapter we develop and test four constructive heuristics for the MCARPTIF with a primary objective to either minimise the total cost, or to minimise the fleet size. The four heuristics tested were adapted from Mixed Capacitated Arc Routing Problem (MCARP) versions and include two *Route-First-Cluster-Second* heuristics that employ the splitting procedures of Chapter 3. Since the MCARPTIF heuristics can be used as-is on the undirected version of the problem, computational tests are also performed on the Capacitated Arc Routing Problem under Time Restrictions with Intermediate Facilities (CARPTIF). We further analysed how the treatment of the vehicle fleet size influenced the performance of the heuristics, and we developed a vehicle reduction procedure that allowed heuristics to better deal with cases where the fleet size have to be minimised. Benchmark tests were performed on realistic waste collection instances, thus improving the practical significance of our results.

In the next section we briefly discuss practical route planning requirements and review constructive heuristics that have been successfully applied for CARPs. In Section 4.3 we present algorithm descriptions of the constructive and vehicle reduction heuristics for the MCARPTIF. In Section 4.4, we describe our heuristic evaluation methods and report computational experiments on the heuristics, focusing on fleet size and solution cost minimisation as well as computation times. The main findings of the tests are discussed in Section 4.5.

4.2 Constructive heuristics for CARPs

The motivation for our research in constructive heuristics stems from a project conducted for a metropolitan municipality in South Africa. The municipality's Waste Collection Department investigated the merits of a proposed recycling programme aimed at job creation. The goal was to increase recycling in the area, shown in Figure 4.1[1], and outsource the collection and transportation process to members of the community. The Department would provide transportation equipment and the community would collect and sell collected material. As further motivation it was argued that recycling would reduce general municipal waste in the areas. We were subsequently tasked with investigating the potential impact of the programme on the Department's municipal waste collection routes. Operational constraints and considerations of the Department were consistent with the

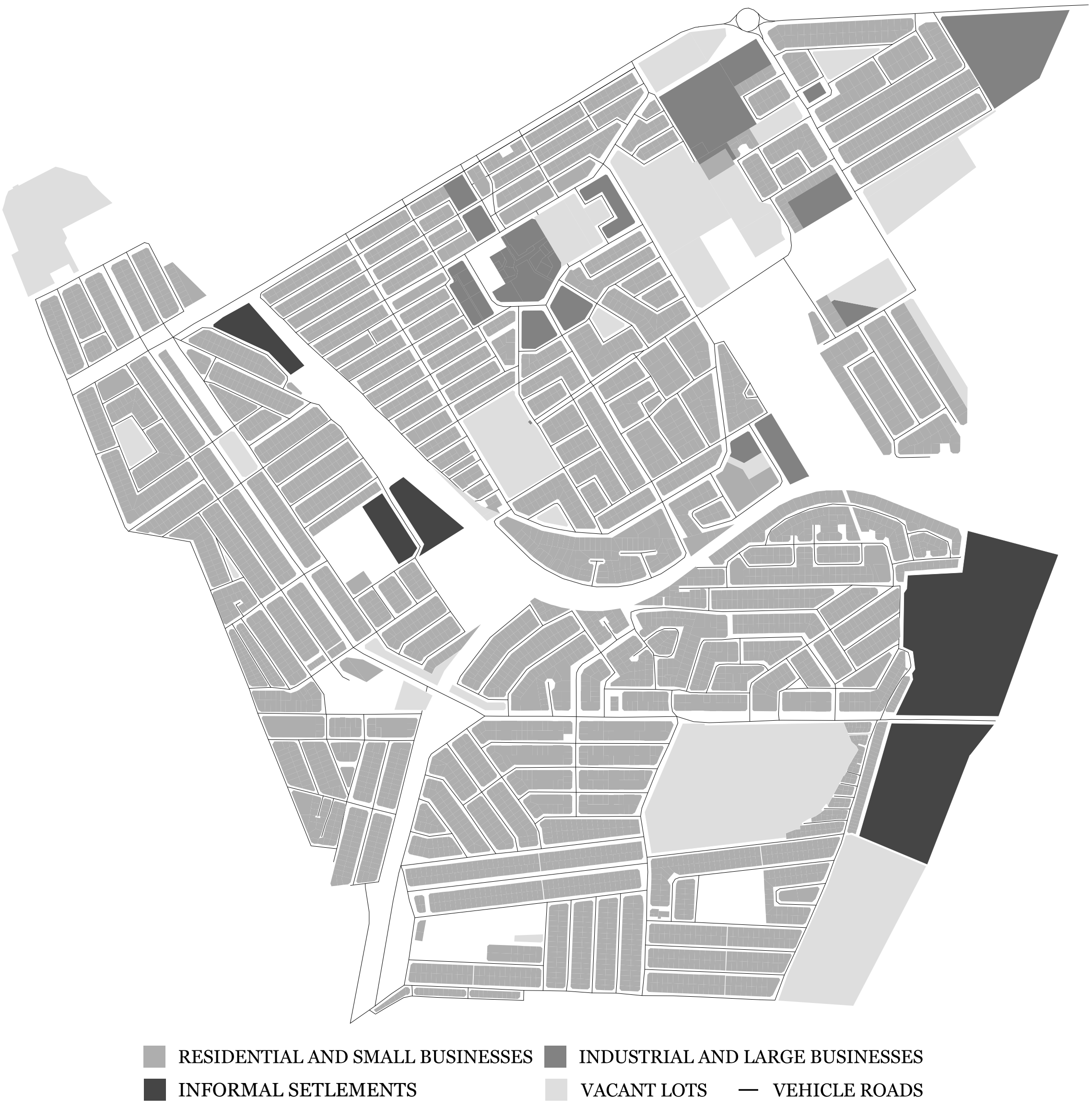

RESIDENTIAL AND SMALL BUSINESSES
INDUSTRIAL AND LARGE BUSINESSES
INFORMAL SETLEMENTS
VACANT LOTS
VEHICLE ROADS

Figure 4.1: Waste collection area for the recycling case study.

[1] The *Act-IF* benchmark set was developed using the case study area and operational data from the municipality.

MCARPTIF, and since possible changes in routes were only one component of the programme, the problem had to be solved in a collaborative real-time planning environment involving stakeholders with different objectives. Scenarios were provided by stakeholders who would use our MCARPTIF solutions as input for evaluating the scenarios, and proposing new ones. Stakeholders would often propose "small" changes to scenarios and expect immediate results. Such changes included using smaller vehicles with less capacity, extending working hours, and including night shifts. The MCARPTIF thus needed to be solved quickly, and repeatedly. These requirements are not unique in waste collection planning, see for example [76], and can be met using constructive heuristics. For the project we identified and adapted the best MCARP heuristics to the MCARPTIF, and it was during our initial review on existing heuristics that we identified the research gaps addressed in this chapter.

Numerous constructive heuristics, which follow simple greedy rules to progressively build approximate feasible solutions, have been proposed for the CARP. Coutinho-Rodrigues et al. [21] compare the performance of classical implementations on the CARP and conclude that *Path-Scanning*, proposed in [39] and one of the most commonly used and studied heuristics, performs the best when CPU time is critical. Beginning with Pearn [67], *Path-Scanning* has been modified by including randomised construct mechanisms. This allows the heuristic to generate and evaluate a number of different solutions for a problem instance, from which the best is returned. Tests by Belenguer et al. [6] show that, on average, such versions need to evaluate between 4 and 21 solutions to match their deterministic versions. A *Path-Scanning* version for the CARP that makes use of an Ellipse-Rule is proposed by Santos et al. [77]; their tests have demonstrated that it performs better than the implementations of Golden and Wong [39] and Pearn [67], but the said *Path-Scanning* version requires fine-tuning of a heuristic parameter. Belenguer et al. [6] compare their *Path-Scanning* versions against their improved version of *Augment-Merge*, first proposed by Golden et al. [38], and *Ulusoy-Partitioning*, a *Route-First-Cluster-Second* heuristic, introduced by Ulusoy [82]. For CARP and MCARP benchmark instances, their *Augment-Merge* version, termed *Improved-Merge*, performed the best, followed by *Ulusoy-Partitioning* and then *Path-Scanning*. To demonstrate the robustness of their metaheuristic on the CARPTIF, Polacek et al. [68] use a random giant route generator and a *Next-Fit* bin-packing heuristic to purposely construct poor starting solutions, which are then improved using the metaheuristic. Lastly, Ghiani et al. [31] developed a *Route-First-Cluster-Second* heuristic to solve the CARPTIF.

In this chapter we extended the *Improved-Merge* and the *Path-Scanning* versions of Belenguer et al. [6] to the MCARPTIF. The heuristics are tested against *Route-First-Cluster-Second* heuristics that employ *Efficient-Split* (Algorithm 3.2) and *Heuristic-Split* (Algorithm 3.3), both introduced in Chapter 3. The aim of the analysis was to identify the best performing constructive heuristics based on their computing time, and ability to find minimum cost and minimum fleet size solutions.

4.3 Constructive heuristics for the MCARPTIF

The following notations, introduced in Section 2.2.4 and summarised in Tables 2.1 and 2.2, are used in the rest of this chapter. An MCARPTIF solution, $\boldsymbol{T}$, is a list, $[\boldsymbol{T}_1, \dots, \boldsymbol{T}_K]$, of K vehicle routes. Each route, $\boldsymbol{T}_i$, is a list of subtrips $[\boldsymbol{T}_{i,1}, \dots, \boldsymbol{T}_{i,|\boldsymbol{T}_i|}]$, and each subtrip, $\boldsymbol{T}_{i,j}$, consists of a list of arc tasks $[T_{i,j,1}, \dots, T_{i,j,|T_{i,j}|}]$. The service time and demand of arc u are given by $w(u)$ and $q(u)$, respectively. A pointer function, $inv(u) = v$, is used to return the opposing arc of u where u and v are the opposing directions of an edge in the original graph, $\boldsymbol{G}$. If u represents an arc in $\boldsymbol{G}$ the pointer returns $inv(u) = 0$.

The travel time for the shortest path from arc u to arc v, which excludes the time of deadheading u and v, is given by $D(u, v)$. It is assumed that the shortest path is always followed between consecutive tasks. The best Intermediate Facility (IF) to visit after servicing arc u and before servicing arc v is pre-calculated and given by $\Phi^*(u, v)$. The duration of the visit, *excluding* offloading time, λ, is given by $\mu^*(u, v)$. It is assumed that the best IF is always visited between two arcs and between an arc and the depot, σ.

For the MCARPTIF the load of a subtrip is given by $load(\boldsymbol{T}_{i,j})$ and it may not exceed Q. The cost of subtrip $\boldsymbol{T}_{i,j}$ is calculated using Equation (4.1), and the cost of the route $\boldsymbol{T}_i$ using Equation (4.2):

$$Z_{\mathrm{S}}(\boldsymbol{T}_{i,j}) = \sum_{n=1}^{|\boldsymbol{T}_{i,j}|-1} \Big(D(T_{i,j,n}, T_{i,j,n+1})\Big) + \sum_{n=1}^{|\boldsymbol{T}_{i,j}|} w(T_{i,j,n}) + \lambda \tag{4.1}$$

$$Z(\boldsymbol{T}_i) = \sum_{j=1}^{|\boldsymbol{T}_i|} Z_{\mathrm{S}}(\boldsymbol{T}_{i,j}). \tag{4.2}$$

In certain cases it is convenient to only include required arcs in $\boldsymbol{T}_i$, which we refer to as a partial route, $\boldsymbol{T}_i^{\mathrm{P}}$. When this is the case, Equation (4.2) will still be used, and the route's full cost will be calculated using Equation (4.3):

$$Z_{\mathrm{P}}(\boldsymbol{T}_i^{\mathrm{P}}) = Z(\boldsymbol{T}_i^{\mathrm{P}}) + D(\sigma, T_{i,1,1}^{\mathrm{P}}) + \sum_{j=2}^{|\boldsymbol{T}_i^{\mathrm{P}}|-1} \mu^*(T_{i,j,|\boldsymbol{T}_{i,j}^{\mathrm{P}}|}^{\mathrm{P}}, T_{i,j+1,1}^{\mathrm{P}}) + \mu^*(T_{i,|\boldsymbol{T}_i^{\mathrm{P}}|,|\boldsymbol{T}_{i,j}^{\mathrm{P}}|}^{\mathrm{P}}, \sigma), \tag{4.3}$$

in which deadheading times to and from the depot and IFs are automatically added. 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, $\boldsymbol{T}_i$, consists of a list of tasks $[T_{i,1}, \ldots, T_{i,|\boldsymbol{T}_i|}]$. Equation (4.1) is also used to calculate the cost of an MCARP route.

4.3.1 Path-Scanning

The first MCARP heuristic that we adapted for the MCARPTIF was *Path-Scanning*. For the MCARP the algorithm starts with a single route containing the depot. The algorithm then iteratively adds the closest required unserviced arc, time-wise, to the end of the current route, while ensuring that there is sufficient remaining capacity on the route to service the arc. If there is insufficient capacity to add any unserviced arc the route is closed by adding a final visit back to the depot. A new route is then opened and the process repeats. When all required arcs have been added to a route, the last open route is closed and the process terminates.

The heuristic deals with mixed networks by evaluating all unserviced arcs and both arc orientations of edges when searching for the closest arc to add to a route. When an arc is added to a solution, its inverse (if it exists) is also marked as serviced. This allows the algorithm to deal with both undirected and mixed networks. Let u be the last arc added to $\boldsymbol{T}_i$ and let $\boldsymbol{R}_s$ be the set of required arcs still requiring service. Further, let

$$\boldsymbol{P}^{(i)} = \{x \in \boldsymbol{R}_s : load(\boldsymbol{T}_i) + q(x) \leq Q\}, \tag{4.4}$$

be the set of arcs that can be added without exceeding vehicle capacity. The next arc, v, to be added to $\boldsymbol{T}_i$ is given by

$$v = \arg\min\{D(u, x) : x \in \boldsymbol{P}^{(i)}\}. \tag{4.5}$$

For the CARP it is common for the heuristic to find multiple unserviced arcs that are at the minimum distance from arc u. These arcs are referred to as the set $\boldsymbol{S}$ of "tied" arcs, given by Equations (4.6) and (4.7):

$$d^* = \min\{D(u,x) : x \in \boldsymbol{P}^{(i)}\}, \tag{4.6}$$

$$\boldsymbol{S} = \{x \in \boldsymbol{P}^{(i)} : D(u,x) = d^*\}. \tag{4.7}$$

Different conditions have been proposed to select the arc to add from $\boldsymbol{S}$, the most popular being the five rules proposed by Golden and Wong [39], which are:

$$\text{Rule 1: } v = \arg\max\left\{D(x,\sigma) : x \in \boldsymbol{S}\right\} \tag{4.8}$$

$$\text{Rule 2: } v = \arg\min\left\{D(x,\sigma) : x \in \boldsymbol{S}\right\} \tag{4.9}$$

$$\text{Rule 3: } v = \arg\max\left\{\frac{w(x)}{q(x)} : x \in \boldsymbol{S}\right\} \tag{4.10}$$

$$\text{Rule 4: } v = \arg\min\left\{\frac{w(x)}{q(x)} : x \in \boldsymbol{S}\right\} \tag{4.11}$$

$$\text{Rule 5: } v = \begin{cases} \text{use Equation (4.8) if } \mathit{load}(\boldsymbol{T}_i) \leq 0.5Q, \\ \text{otherwise use Equation (4.9).} \end{cases} \tag{4.12}$$

Rules 1 and 2 respectively maximise and minimise the time to the depot, and Rules 3 and 4 respectively maximise and minimise the arc yield (service time divided by demand). Rule 5 forces the addition of arcs that are further away from the depot when the vehicle is less than half-empty, and closer to the depot otherwise. The algorithm is executed five times using a different rule for each run to choose v from $\boldsymbol{S}$, and the best solution from the five runs is chosen. Pearn [67] considers a random tie-break scheme, whereby a rule is uniformly chosen at random whenever $|\boldsymbol{S}| > 1$. The advantage of this approach is that an arbitrary number of solutions can be generated, and the best returned. Belenguer et al. [6] propose a rule-free version, termed *Path-Scanning-Random-Link*, whereby the arc to add is uniformly chosen at random from $\boldsymbol{S}$. This approach also allows for an arbitrary number of solutions to be generated, and the best chosen.

Path-Scanning for the MCARPTIF

The MCARPTIF sees the introduction of subtrips, $\boldsymbol{T}_{i,j}$, and the demand collected on a subtrip cannot exceed Q. The duration of a route cannot exceed L. Let u be the last arc added to subtrip $\boldsymbol{T}_{i,j}$ and let $x \in \boldsymbol{R}_s$ be a candidate arc still requiring service. The algorithm has to check for both the capacity limit on a subtrip and the duration of a route. As a result three sets are defined. The first set,

$$\boldsymbol{P}^{(i)} = \{k \in \boldsymbol{R}_s : Z(\boldsymbol{T}_i) + D(u,x) + w(x) + \mu^*(x,\sigma) \leq L\}, \tag{4.13}$$

consists of arcs that can be added to the route without exceeding L. If $\boldsymbol{P}^{(i)} = \varnothing$, meaning it is an empty set, the route is full as the time of servicing any arc x and returning to the final IF and depot visit exceeds L. The second set,

$$\boldsymbol{P}^{(i,j)} = \{x \in \boldsymbol{P}^{(i)} : \mathit{load}(\boldsymbol{T}_{i,j}) + q(x) \leq Q\}, \tag{4.14}$$

is a subset of $\boldsymbol{P}^{(i)}$ and consists of arcs that can be added to the current subtrip without exceeding Q. The third set,

$$\boldsymbol{P}^{(i)}_{\text{IF}} = \{x \in \boldsymbol{P}^{(i)} : Z(\boldsymbol{T}_i) + \mu^*(u,x) + w(x) + \mu^*(x,\sigma) + \lambda \leq L\}, \tag{4.15}$$

consists of arcs that can be added to a route after an IF visit, hence in a new subtrip, without exceeding L. If both $\boldsymbol{P}^{(i,j)} = \varnothing$ and $\boldsymbol{P}_{\text{IF}}^{(i)} = \varnothing$, the route is full. The arc from $\boldsymbol{S}$ to be added to subtrip $\boldsymbol{T}_{i,j}$ is given by:

$$d^* = \begin{cases} \min\{D(u,x) : x \in \boldsymbol{P}^{(i,j)}\} & \text{if } \boldsymbol{P}^{(i,j)} \neq \varnothing, \\ \infty & \text{otherwise,} \end{cases} \tag{4.16}$$

$$\boldsymbol{S} = \{x \in \boldsymbol{P}^{(i,j)} : D(u,x) = d^*\}, \tag{4.17}$$

and the next arc from $\boldsymbol{S}_{\text{IF}}$ to be added to a new subtrip $T_{i,j+1}$ is given by:

$$d_{\text{IF}}^* = \begin{cases} \min\{\mu^*(u,x) : x \in \boldsymbol{P}_{\text{IF}}^{(i)}\} & \text{if } \boldsymbol{P}_{\text{IF}}^{(i)} \neq \varnothing, \\ \infty & \text{otherwise,} \end{cases} \tag{4.18}$$

$$\boldsymbol{S}_{\text{IF}} = \{x \in \boldsymbol{P}_{\text{IF}}^{(i)} : \mu^*(u,x) = d_{\text{IF}}^*\}. \tag{4.19}$$

If $\boldsymbol{P}_{\text{IF}}^{(i)} \neq \varnothing$ and $d_{\text{IF}}^* < d^*$, it is cheaper to visit an IF and add an unserviced arc than to directly add an unserviced arc to the current subtrip. This requires the heuristic to find and compare d^* and d_{IF}^*, which adds to its computational complexity. A more efficient version can be used that only calculates $\boldsymbol{P}_{\text{IF}}^{(i)}$ if $\boldsymbol{P}^{(i,j)} = \varnothing$. During our preliminary tests we found both the efficient and full versions to perform similarly and limit our results to the efficient version.

To allow the heuristic to deal with multiple closest arcs, the tie-break rules can be updated since vehicles have to visit IFs. Santos et al. [76] update the five rules by using the travel time from the candidate arc to the IF instead of the travel time to the depot. This modification is only possible when the problem has one IF. For the case with multiple IFs we propose that the travel time from the candidate arc to the *nearest* IF should be used. Furthermore, the original rules involving depot travel times are still valid for the MCARPTIF and need not be replaced. Instead, rules involving IF travel times are considered as additional rules and formally defined as follows:

$$\text{Rule 6: } v = \arg\max\left\{D(x,y) : x \in \boldsymbol{S}, y \in \boldsymbol{I}\right\} \tag{4.20}$$

$$\text{Rule 7: } v = \arg\min\left\{D(x,y) : x \in \boldsymbol{S}, y \in \boldsymbol{I}\right\} \tag{4.21}$$

An eighth rule that we propose takes into account L, whereby arcs that are closer to the last IF and depot visit are chosen when the route is at least 75% of its route duration restriction. Otherwise if the subtrip is more than half full, arcs closer to the nearest IF are chosen:

$$\text{Rule 8: } v = \begin{cases} \arg\min\left\{\mu^*(x,\sigma) : x \in \boldsymbol{S}\right\} \text{ if } Z(\boldsymbol{T}_i) \geq 0.75L, \\ \text{otherwise use Equation (4.20) if } \mathit{load}(\boldsymbol{T}_{i,j}) \leq 0.5Q, \\ \text{otherwise use Equation (4.21).} \end{cases} \tag{4.22}$$

For the deterministic version, one of the eight rules can be applied per run, and the best solution over the eight runs returned. Preliminary tests found none of the rules to be redundant in that each rule resulted in the best solution for at least a few problem instances. For the randomised version, one of the eight rules or of a subset of the eight rules can be chosen at random. The simplest option is to avoid using any rules and choose a closest arc at random, in the same manner as for *Path-Scanning-Random-Link* for the MCARP.

Since *Path-Scanning* produces different solutions per run it is embedded in a high-level procedure, *nConstruct* (Algorithm 4.1, shown at the end of the chapter), which generates α solutions and returns the best fleet size solution, the best route cost solution, and the best route cost solution while adhering to the fleet limit if it is applied. A detailed description of *Path-Scanning* generating an MCARPTIF solution is shown at the end of the chapter in Algorithm 4.2. In terms of computational complexity, all versions of *Path-Scanning* can generate a solution within $\mathcal{O}(\tau^2)$, where $\tau = |\boldsymbol{R}|$, making it efficient even when dealing with large problem instances.

This chapter presents results for two versions of *Path-Scanning*. The first is the deterministic version completing eight runs, each linked with one of the eight tie-break rules and returning the least cost and least fleet size solutions. The second is the randomised rule-free version that chooses a closest arc at random and completes a user-specified number of runs, and returns the best solutions found for the two primary objectives. Tests were also conducted on random-rule based versions, which included a version that chooses from Rules 1 to 5, and a version that chooses from Rules 1, 2 and 6 to 8. Both randomised versions performed similarly to the *Path-Scanning-Random-Link* version. We report only on results for *Path-Scanning-Random-Link* since it is the easiest to implement.

4.3.2 Merge

The second heuristic that we modified for the MCARPTIF was *Improved-Merge*, developed by Belenguer et al. [6] for the MCARP. Their algorithm is a simplified, yet an improved version of *Augment-Merge* [38]. For the MCARPTIF it consists of two steps:

Step 1 : For each required arc, construct a feasible route servicing the arc, starting at the depot, and ending with the best IF visit followed by the depot. Where two arcs are opposing and represent the same edge task, only include the route servicing the edge in its best orientation.

Step 2 : Subject to subtrip capacity and route duration restrictions, evaluate the merge of any two routes. Merge the two routes which yield the best saving, and repeat Step 2. If no feasible merge can be found, stop.

Using partial routes, that is without dummy arcs representing the depot or IFs, two routes, $\boldsymbol{T}_i^{\mathrm{P}}$ and $\boldsymbol{T}_j^{\mathrm{P}}$, can be directly merged by combining the last subtrip of $\boldsymbol{T}_i^{\mathrm{P}}$ with the first subtrip in $\boldsymbol{T}_j^{\mathrm{P}}$. We refer to such a merge of the routes $\boldsymbol{T}_i^{\mathrm{P}}$ and $\boldsymbol{T}_j^{\mathrm{P}}$ as $i \rightarrow j$. If $\boldsymbol{T}_i^{\mathrm{P}} = [[a],[b,u]]$ and $\boldsymbol{T}_j^{\mathrm{P}} = [[v,x]]$, their merge would result in $\boldsymbol{T}_{i\rightarrow j}^{\mathrm{P}} = \big[[a],[b,u,v,x]\big]$, subject to the demand of the combined subtrips not exceeding Q, and the duration of the merged routes not exceeding L. The merge saving, m, is:

$$m = \mu^*(u,\sigma) + D(\sigma,v) + \lambda - D(u,v), \tag{4.23}$$

which is the positive travel time saving of going directly from arc u to v, instead of going from u to an IF, incurring offload time, going to the depot, and the next route going from the depot to arc v. Routes can also be merged through best IF visits, resulting in $\boldsymbol{T}_{i\Rightarrow j}^{\mathrm{P}} = \big[[a],[b,u],[v,x]\big]$, which is only subject to the duration of the merged routes not exceeding L. We refer to such a merge between routes $\boldsymbol{T}_i^{\mathrm{P}}$ and $\boldsymbol{T}_j^{\mathrm{P}}$ as $i \Rightarrow j$. For this case, the positive merge saving is:

$$m_{\mathrm{IF}} = \mu^*(u,\sigma) + D(\sigma,v) - \mu^*(u,v). \tag{4.24}$$

Since the CARP considers an undirected network, CARP routes are symmetrical, meaning route $\boldsymbol{T}_i^{\mathrm{P}} = [u, v, x]$ has the same cost as $\boldsymbol{T}_{i'}^{\mathrm{P}} = [inv(x), inv(v), inv(u)]$, and $j' \to i'$ gives the same saving as $i \to j$. This results in there being an additional two unique mergers: $i \to j'$ and $i' \to j$, where i' is the inverse of route i. These four mergers capture all possible savings between two CARP routes. The same does not apply to the MCARP, since not all arc tasks have inverse tasks, and even if they do, the route is only symmetrical if the deadheading times between these tasks are the same in both directions. To keep *Merge* efficient, Belenguer et al. [6] propose that mergers involving i' and j' only be considered if both routes $\boldsymbol{T}_i^{\mathrm{P}}$ and $\boldsymbol{T}_j^{\mathrm{P}}$ are symmetrical and the new deadheading path linking the routes is also symmetrical. If this condition is not enforced, all eight merge orientations will have to be calculated. We propose the same for the MCARPTIF and the CARPTIF, even though the latter is undirected. For the CARPTIF, the first subtrip in a route starts with $[\sigma, u, \ldots]$ and its last subtrip ends with $[\ldots, v, \Phi_i, \sigma]$, so unless $D(\sigma, inv(v)) = \mu^*(inv(u), \sigma)$ the route is asymmetrical. When merging two routes, this condition must also hold between the first arc in route $\boldsymbol{T}_i^{\mathrm{P}}$ and the last arc in route $\boldsymbol{T}_j^{\mathrm{P}}$, otherwise mergers involving their inverse will have different savings.

Similar to *Path-Scanning*, it may happen that different route mergers have equal savings. For the MCARP, Belenguer et al. [6] propose that a route demand discrepancy rule be used to break ties whereby $\boldsymbol{T}_i^{\mathrm{P}}$ and $\boldsymbol{T}_j^{\mathrm{P}}$ are chosen to maximise $|load(\boldsymbol{T}_i^{\mathrm{P}}) - load(\boldsymbol{T}_j^{\mathrm{P}})|$. This version of merge is referred to as *Improved-Merge*. For the MCARPTIF the rule can be used when breaking ties between direct mergers such that $|load(\boldsymbol{T}_{i,|\boldsymbol{T}_i^{\mathrm{P}}|}^{\mathrm{P}}) - load(\boldsymbol{T}_{j,1}^{\mathrm{P}})|$ is maximised. A cost discrepancy rule, maximising $|Z_{\mathrm{P}}(\boldsymbol{T}_i^{\mathrm{P}}) - Z_{\mathrm{P}}(\boldsymbol{T}_j^{\mathrm{P}})|$, can be used to break ties between mergers through IFs. An alternative is to only use the cost discrepancy rule; and we found this version to dominate the hybrid load and cost discrepancy rule. Similar to *Path-Scanning*, another option for the MCARP and MCARPTIF is to break-ties by randomly choosing a merger. We refer to this version as *Randomised-Merge*. It has the ability to produce different solutions over multiple runs, from which the best can be returned. Multiple runs do require more computational time, and more efficient implementations become critical, especially when dealing with realistically sized instances. Belenguer et al. [6], who implemented an $\mathcal{O}(\tau^3)$ version where $\tau = |\boldsymbol{R}|$, following the structure just described, note that *Improved-Merge* can be implemented in a non-trivial way in $\mathcal{O}(\tau^2 \log \tau)$. In this chapter we present such an implementation.

Efficient Merge implementation

Referring to Equations (4.23) and (4.24), the merge savings of $i \to j$ and $i \Rightarrow j$ only depend on the last arc in subtrip $\boldsymbol{T}_{i,|\boldsymbol{T}_i^{\mathrm{P}}|}^{\mathrm{P}}$ and the first arc in subtrip $\boldsymbol{T}_{j,1}^{\mathrm{P}}$. Merge savings between all arcs are calculated in the first merge phase, and the savings remain the same in subsequent phases. Merge savings thus only have to be calculated once between all arcs and stored in a merge savings list, $\boldsymbol{M} = [M_1, .., M_{|\boldsymbol{M}|}]$. The savings list is then sorted in nonincreasing order, and starting with the first entry, $m = M_1$, the merge can be implemented if it is feasible, otherwise the next merge in the list is evaluated. This repeats until the entire list has been scanned. For the efficient implementation, we again use partial routes without dummy arcs, and define mergers between arcs instead of routes, meaning $u \to v$ is the direct merge of arcs u and v, and $u \Rightarrow v$ is the merge of arcs u and v via an IF visit.

Step 1: Initialisation For the first step, a route with one subtrip is constructed for each required arc:

$$\boldsymbol{T}^{\mathrm{P}} = \Big[[[u]] \quad \forall \quad u \in \boldsymbol{R}\Big]. \tag{4.25}$$

We define $T^{-1}(u)$ as a dynamic pointer function, mapping arc u to its route's index i, which is updated after each merge:

$$T^{-1}(u) = \begin{cases} i \in \{1, .., |\boldsymbol{T}^{\mathrm{P}}|\} \text{ if } u \in \{\boldsymbol{T}^{\mathrm{P}}_{i,1,1}\} \cup \{\boldsymbol{T}^{\mathrm{P}}_{i,|\boldsymbol{T}^{\mathrm{P}}_i|,|\boldsymbol{T}^{\mathrm{P}}_{i,|\boldsymbol{T}^{\mathrm{P}}_i|}|}\}, \\ 0 \text{ otherwise,} \end{cases} \tag{4.26}$$

A merge can only involve the last arc in $\boldsymbol{T}^{\mathrm{P}}_i$ and the first arc in $\boldsymbol{T}^{\mathrm{P}}_j$. The rest of the arcs in the route are "locked-in". When $inv(u) \neq 0$, both u and $inv(u)$ are included in $\boldsymbol{T}^{\mathrm{P}}$ by Equation (4.25), instead of including only the best orientation. The reason is that mergers involving bad orientations may still be good enough to overcome extra costs incurred by the orientation. To check if such mergers are indeed better, a penalty function is defined:

$$z(u) = Z_{\mathrm{P}}\big(\boldsymbol{T}^{\mathrm{P}}_{T^{-1}(u)}\big), \tag{4.27}$$

$$pen(u) = \begin{cases} z(u) - z\big(inv(u)\big) & : inv(u) \neq 0, z(u) > z\big(inv(u)\big) \\ 0 & \quad \text{otherwise,} \end{cases} \tag{4.28}$$

$$p(u,v) = pen(u) + pen(v), \tag{4.29}$$

For convenience, Equation (4.27) defines a cost function, $z(u)$, that gives the cost of an arc's route. The penalty $p(u,v)$, given by Equations (4.28) and (4.29), is to be subtracted from the merge saving between arcs u and v. The merge savings list is calculated using Equations (4.30) to (4.32), which consist of mergers resulting from $u \rightarrow v$ and from $u \Rightarrow v$:

$$\begin{aligned} \boldsymbol{M}_{\rightarrow} =& \big[\mu^*(u,\sigma) + D(\sigma,v) + \lambda - D(u,v) - p(u,v) \\ & \quad \forall\, (u,v) : u, v \in \boldsymbol{R}, v \notin \{u, inv(u)\}\big], \end{aligned} \tag{4.30}$$

$$\begin{aligned} \boldsymbol{M}_{\Rightarrow} =& \big[\mu^*(u,\sigma) + D(\sigma,v) - \mu^*(u,v) - p(u,v) \\ & \quad \forall\, (u,v) : u, v \in \boldsymbol{R}, v \notin \{u, inv(u)\}\big], \end{aligned} \tag{4.31}$$

$$\boldsymbol{M} = \boldsymbol{M}_{\rightarrow} \cup \boldsymbol{M}_{\Rightarrow}, \tag{4.32}$$

Two pointer functions, $\boldsymbol{M}^{-1}_{\rightarrow}(m)$ and $\boldsymbol{M}^{-1}_{\Rightarrow}(m)$, are used to map the merge saving, m, to the set of arc pairs, (u,v), that results in the saving:

$$\boldsymbol{M}^{-1}_{\rightarrow}(m) = \begin{cases} \{(u,v) : \mu^*(u,\sigma) + D(\sigma,v) + \lambda - D(u,v) - p(u,v) = m\} \\ \varnothing \text{ otherwise,} \end{cases} \tag{4.33}$$

$$\boldsymbol{M}^{-1}_{\Rightarrow}(m) = \begin{cases} \{(u,v) : \mu^*(u,\sigma) + D(\sigma,v) - \mu^*(u,v) - p(u,v) = m\} \\ \varnothing \text{ otherwise.} \end{cases} \tag{4.34}$$

After populating $\boldsymbol{M}$ with all the merge savings, the duplicate savings are removed, the list is sorted in nonincreasing order, and $n = 1$ so that $m = M_1$ is the first merge evaluated in Step 2 of the heuristic. The mappings return all the arc pairs in the case of tied-mergers with equal savings; hence why duplicates can be removed from $\boldsymbol{M}$.

Step 2: Feasible Merge-Identification For the second step, let $m = M_n$. A merge between any two arcs u and v involving routes $i = T^{-1}(u)$ and $j = T^{-1}(v)$ is only feasible if the arc is still in the solution (eq. (4.35)), arc u is the last arc in its route and arc v the first (eq. (4.36)), the merge does not involve the same routes (eqs. (4.37) and (4.38)), and the cost of the merged routes does not exceed the route duration restriction (eq. (4.39)):

$$i \neq 0 \text{ and } j \neq 0, \tag{4.35}$$

$$u = \boldsymbol{T}^{\mathrm{P}}_{i,|\boldsymbol{T}^{\mathrm{P}}_i|,|\boldsymbol{T}^{\mathrm{P}}_{i,|\boldsymbol{T}^{\mathrm{P}}|}|} \text{ and } v = \boldsymbol{T}^{\mathrm{P}}_{j,1,1}, \tag{4.36}$$

$$i \neq j, \tag{4.37}$$

$$i \neq T^{-1}\big(inv(v)\big) : inv(v) \neq 0 \text{ and } T^{-1}\big(inv(v)\big) \neq 0, \tag{4.38}$$

$$z(u) + z(v) - m - p(u,v) \leq L. \tag{4.39}$$

Mergers may be disqualified by Equation (4.35) if their opposing arcs are in asymmetrical routes which are the product of previous mergers. Equation (4.36) checks that arcs u and v are not already "locked-in". If mergers are allowed with inverted symmetrical routes, Equation (4.38) ensures that a symmetrical route is not merged with its inverted self. In Equation (4.39) the orientation penalties are subtracted since they are not actually incurred in a route. If a merge is to be implemented as $u \to v$, it is only feasible if the demand of the merged subtrips does not exceed vehicle capacity:

$$load\big(\boldsymbol{T}^{\mathrm{P}}_{i,|\boldsymbol{T}^{\mathrm{P}}_i|}\big) + load\big(\boldsymbol{T}^{\mathrm{P}}_{j,1}\big) \leq Q. \tag{4.40}$$

The sets of feasible mergers, $\boldsymbol{M}^{-1}_{\rightarrow}(m)$ and $\boldsymbol{M}^{-1}_{\Rightarrow}(m)$, are reduced to $\boldsymbol{F}_{\rightarrow}$ and $\boldsymbol{F}_{\Rightarrow}$ that contain only feasible mergers:

$$\boldsymbol{F}_{\rightarrow} = \{(u,v) \in \boldsymbol{M}^{-1}_{\rightarrow}(m) : \text{ eqs. (4.37) to (4.40)}\}, \tag{4.41}$$

$$\boldsymbol{F}_{\Rightarrow} = \{(u,v) \in \boldsymbol{M}^{-1}_{\Rightarrow}(m) : \text{ eqs. (4.37) to (4.39)}\}. \tag{4.42}$$

Step 3: Merge-Implementation If $\boldsymbol{F}_{\rightarrow} = \varnothing$ and $\boldsymbol{F}_{\Rightarrow} = \varnothing$, there are no feasible mergers to implement with savings M_n, in which case $n = n+1$ and the heuristic returns to *Step 2*, unless $n > |\boldsymbol{M}|$, which means that all mergers have been evaluated and the heuristic terminates.

If there are feasible mergers to implement, different tie-break rules can then be used to choose pair (u,v) from $\boldsymbol{F}_{\rightarrow}$ or $\boldsymbol{F}_{\Rightarrow}$, such as the route cost discrepancy rule:

$$(u,v) = \arg\max\big\{|z(u) - z(v)| : (u,v) \in \boldsymbol{F}_{\rightarrow} \cup \boldsymbol{F}_{\Rightarrow}\big\}, \tag{4.43}$$

or a pair can be chosen at random. If (u,v) is chosen from $\boldsymbol{M}^{-1}_{\rightarrow}(m)$, let $i = T^{-1}(u)$ and $j = T^{-1}(v)$. The merge is then implemented using Equations (4.44) to (4.46):

$$\boldsymbol{T}^{\mathrm{P}}_{i,|\boldsymbol{T}^{\mathrm{P}}_i|} = \boldsymbol{T}^{\mathrm{P}}_{i,|\boldsymbol{T}^{\mathrm{P}}_i|} \cup \boldsymbol{T}^{\mathrm{P}}_{j,1}, \tag{4.44}$$

$$\boldsymbol{T}^{\mathrm{P}}_{i} = \boldsymbol{T}^{\mathrm{P}}_{i} \cup \boldsymbol{T}^{\mathrm{P}}_{j} \setminus \boldsymbol{T}^{\mathrm{P}}_{j,1}, \tag{4.45}$$

$$\boldsymbol{T}^{\mathrm{P}}_{j} = \varnothing. \tag{4.46}$$

In Equation (4.44), the last subtrip in $\boldsymbol{T}^{\mathrm{P}}_{i}$ is combined with the first subtrip of $\boldsymbol{T}^{\mathrm{P}}_{j}$, and in Equation (4.45), $\boldsymbol{T}^{\mathrm{P}}_{i}$ is combined with $\boldsymbol{T}^{\mathrm{P}}_{j}$, excluding its first subtrip whose arcs have already been incorporated in $\boldsymbol{T}^{\mathrm{P}}_{i}$. Since all arcs in $\boldsymbol{T}^{\mathrm{P}}_{j}$ are now serviced in $\boldsymbol{T}^{\mathrm{P}}_{i}$, Equation (4.46) removes $\boldsymbol{T}^{\mathrm{P}}_{j}$ from solution $\boldsymbol{T}^{\mathrm{P}}$. If (u,v) is chosen from $\boldsymbol{M}^{-1}_{\Rightarrow}(m)$, the merge is implemented as:

$$\boldsymbol{T}^{\mathrm{P}}_{i} = \boldsymbol{T}^{\mathrm{P}}_{i} \cup \boldsymbol{T}^{\mathrm{P}}_{j}. \tag{4.47}$$

In Equation (4.47), the route $\boldsymbol{T}_i^{\mathrm{P}}$ is combined with $\boldsymbol{T}_j^{\mathrm{P}}$, without merging any subtrips, and Equation (4.46) is applied thereafter. After implementing a merge, arcs u and v are "locked-in", and $T^{-1}(u) = 0$ and $T^{-1}(v) = 0$. Additional steps are required depending on whether inverting symmetrical routes are allowed, as discussed next. After a merge is implemented the heuristic returns to Step 3.

Inverting symmetrical routes *Improved-Merge* and *Randomised-Merge* allow for inverted symmetrical routes to also be considered for a merge. As part of Step 1, $\boldsymbol{S'}$ is defined as the set that initially contains all arcs that are part of symmetrical routes, such that:

$$\boldsymbol{S'} = \{u \in \boldsymbol{R} : inv(u) \neq 0, pen(u) = 0\}. \tag{4.48}$$

In Step 3, the route resulting from $u \rightarrow v$ or from $u \Rightarrow v$ is symmetrical if Equations (4.49) and (4.50), or Equations (4.49) and (4.51) hold, respectively:

$$u, v \in \boldsymbol{S'}, \tag{4.49}$$

$$(inv(v), inv(u)) \in \boldsymbol{M}_{\rightarrow}^{-1}(m), \tag{4.50}$$

$$(inv(v), inv(u)) \in \boldsymbol{M}_{\Rightarrow}^{-1}(m). \tag{4.51}$$

After implementing a merge, should these conditions hold, $inv(v) \rightarrow inv(u)$ will also be implemented using Equations (4.44) to (4.46), or $inv(v) \Rightarrow inv(u)$ using Equation (4.47) and (4.46). If these conditions do not hold, and subject to $inv(u) \neq 0$ then:

$$\boldsymbol{T}^{\mathrm{P}}_{T^{-1}(inv(u))} = \varnothing : inv(u) \neq 0, T^{-1}\big(inv(u)\big) \neq 0, \tag{4.52}$$

$$\boldsymbol{T}^{\mathrm{P}}_{T^{-1}(inv(v))} = \varnothing : inv(v) \neq 0, T^{-1}\big(inv(v)\big) \neq 0, \tag{4.53}$$

$$\boldsymbol{S'} = \boldsymbol{S'} \setminus \{u, v\}, \tag{4.54}$$

and $T^{-1}\big(inv(u)\big) = 0$ and $T^{-1}\big(inv(v)\big) = 0$, should arcs u and v have opposites. After all mergers have been evaluated and a final route is symmetrical, both its orientations will be in $\boldsymbol{T}^{\mathrm{P}}$ and any one can be removed. Also, if $u \in \boldsymbol{S'}$ could not be merged with any route, there will be two single arc routes in $\boldsymbol{T}$ for u and $inv(u)$, and the worst of the two routes should be removed. If mergers involving inverted symmetrical routes are not allowed, Equations (4.52) to (4.54) will be automatically applied, regardless of the outcomes of Equations (4.49) to (4.51). A detailed algorithm description of the efficient version of *Randomised-Merge*, with inverted symmetrical route mergers not allowed, is presented in Algorithms 4.3 and 4.4, at the end of the chapter.

In Step 1, calculating all possible mergers takes $\mathcal{O}(2\tau^2)$, and sorting $\boldsymbol{M}$ takes $\mathcal{O}(\tau \log(\tau))$. In Steps 2 and 3, evaluating all possible mergers takes $\mathcal{O}(\tau^2)$. The worst case performance of *Improved-Merge* and *Randomised-Merge* for the MCARPTIF is thus $\mathcal{O}(\tau^2)$, same as *Path-Scanning*. An advantage of the efficient *Merge* versions is that $\boldsymbol{M}$ need only be calculated once. Thereafter Steps 2 to 4 will produce a new solution each time they are applied. Steps 2 and 3 are therefore linked with *nConstruct* (Algorithm 4.1), and the sorted $\boldsymbol{M}$ list is supplied as input data. For our computational tests we used the efficient *Merge* implementation that allowed for symmetrical routes to be inverted. We tested *Improved-Merge* using the cost discrepancy rule as well as *Randomised-Merge*.

4.3.3 Route-First-Cluster-Second heuristics

A *Route-First-Cluster-Second* heuristic was first proposed for the CARP by Ulusoy [82], and was extended to the CARP with IFs by Ghiani et al. [32], to the CARPTIF by Ghiani et al. [31], to the MCARP by Lacomme et al. [51], and to the MCARPTIF in Chapter 3. For the CARP, the heuristic generates a giant tour that services all the required arcs and edges, and partitions the tour via a splitting procedure into feasible smaller routes so as to adhere to the capacity limit. For the CARPTIF and MCARPTIF, the giant tour is optimally partitioned into feasible routes so as to adhere to a route duration restriction, and also into subtrips with IF visits to adhere to the capacity limit. Finding and partitioning the least cost giant tour, which is $\mathscr{NP}$-Hard since it involves the Rural Postman Problem, does not guarantee an optimal final solution. As a result, heuristics are used for the construction of the giant tour. Lacomme et al. [51] and Belenguer et al. [6] relax vehicle capacity of the different *Path-Scanning* versions to generate giant tours. *Improved-Merge* and *Randomised-Merge* can also be used in this fashion. As proposed by Belenguer et al. [6], randomised giant tours constructors can be used to enable the heuristic to partitioning a different solution with each run, from which the best is returned.

For our computational tests we used *Path-Scanning* with six rules and *Path-Scanning-Random-Link* to generate initial giant tours. The *Path-Scanning* rules involving subtrip load and route duration restrictions are redundant since the limits are relaxed by setting $Q \leftarrow \infty$ and $L \leftarrow \infty$. After generating a giant tour it was partitioned using either the optimal splitting procedure (Algorithm 3.2) or the quicker near-optimal splitting procedure (Algorithm 3.3), both introduced in Chapter 3. The implementations are referred to as *Route-Cluster* and *Efficient-Route-Cluster*, respectively[2]. The heuristics were further adapted to either minimise the total cost or number of required vehicles, depending on the main and secondary problem objectives. We also tested the splitting procedures linked with a *Randomised-Merge* giant route generator, but found that it was always dominated by other heuristics and do not report on its performance.

In Chapter 3, preliminary results showed that *Route-Cluster* performed only marginally better than *Efficient-Route-Cluster* when both were allowed the same execution time. However, *Route-Cluster* is more difficult to implement. For this reason we favoured the *Efficient-Route-Cluster* implementation when performing computational tests. Computational results are thus mainly focussed on this implementation.

4.3.4 Vehicle Reduction heuristic

To enable constructive heuristics to better cope with the minimise fleet size objective an efficient vehicle reduction heuristic was developed, termed *Reduce-Vehicles*. The heuristic takes a solution as input, removes a route and using remove-insert operators, attempts to reinsert its arcs in the remaining routes in an effort to reduce the number of vehicles, K.

The heuristic starts by sorting the solution's routes according to their cost. The lowest cost route is removed and its arcs sorted according to individual demand. Starting with the highest demand arc, the algorithm finds the least cost insert position of the arc into one of the remaining route's subtrips, subject to time and capacity limits. If a feasible insert position is found, the arc is added to the route and the next arc evaluated. If an insert position cannot be found, the algorithm reverts back to the starting solution. The process is again applied to the original solution, but starting with the second lowest cost route. If a route is successfully removed, the whole process is repeated on the new reduced solution. The process terminates when none of the routes could be successfully removed. Removal and insertion procedures used by *Reduce-Vehicles* are similar to a *relocate* neighbourhood

[2] Admittedly, the terms may be confusing since *Route-Cluster* uses *Efficient-Split*, and *Efficient-Route-Cluster* uses *Heuristic-Split*. We chose to name the *Heuristic-Split* version *Efficient-Route-Cluster* since it is quicker than *Route-Cluster*. *Route-Cluster* is also based on the standard *Route-First-Cluster-Second* versions that rely on optimal splitting, which in this case is provided by *Efficient-Split*.

search, presented in detail in the next chapter, which has a computational complexity of $\mathcal{O}(\tau^2)$, but since it stops reinserting a route's arcs when a feasible position cannot be found, and arcs are sorted in increasing order, its observed complexity is usually less. A detailed description of *Reduce-Vehicles* can be found in Algorithms 4.5 and 4.6, at the end of the chapter.

4.4 Computational results

The aim of this chapter was to develop and evaluate heuristics for the MCARPTIF in an effort to identify the best heuristic for implementation purposes, to measure the impact of minimising fleet size on heuristic performance, and to determine how performance is further influenced by the choice of benchmark sets. Computational tests were performed on the modified MCARPTIF versions of *Path-Scanning* (PS), *Path-Scanning-Random-Link* (PS-RL), *Improved-Merge* (IM), *Randomised-Merge* (RM), as well as *Efficient-Route-Cluster* (ERC) linked with PS and PS-RL giant route generators, and the optimal partitioning version of *Route-Cluster* (RC) linked with the same giant route generators. In this section, ERC and RC refer to the deterministic versions linked with PS, and the versions linked with PS-RL are labelled RC-RL and ERC-RL. Each heuristic was evaluated in terms of minimising solution cost and minimising fleet size. For the latter we evaluated heuristics when linked with *Reduce-Vehicles*. In an effort to improve the practical value of our results, heuristics were tested on the realistic *Cen-IF* and *Act-IF* waste collection benchmark instances. Tests were also performed on the *Lpr-IF* and *mval-IF-3L* MCARPTIF instances, and on the *gdb-IF*, *gdb-IF-3L*, *bccm-IF* and *bccm-IF-3L* CARPTIF instances. The benchmark instances are discussed in detail in Chapter 2, Section 2.4.2.

Constructive heuristics were programmed in Python version 2.7, with critical procedures optimised using Cython version 0.17.1. Computational experiments were run on a Dell PowerEdge R910 4U Rack Server with 128GB RAM with four Intel Xeon E7540 processors each having 6 cores, and 12 threads and with a 2GHz base frequency. Experiments were run without using programmatic multi-threading or multiple processors.

4.4.1 Evaluation criteria

Five criteria were used to evaluate heuristic performance. The first three were solution cost, the required fleet size of a solution, and computational time required by a heuristic to generate a solution. All randomised heuristics are allowed α runs, from which the best is returned. We refer to such a heuristic execution as RM(α), using *Randomised-Merge* as an example. Two additional criteria were used for randomised heuristic versions, namely the number of runs required for randomised heuristics to break-even with their deterministic versions, and lastly, statistical cost and fleet size intervals for heuristics when allowed a certain number of runs.

With no lower bounds currently available for the MCARPTIF, the least cost and least vehicle fleet size solutions found during all computational tests on constructive heuristics were used as a baseline. Solution quality was measured as the cost gap, Z_{gap}, for a solution $\boldsymbol{T}$, calculated as

$$Z_{\text{gap}} = \frac{\sum_{i=1}^{|\boldsymbol{T}|} Z(\boldsymbol{T}_i) - Z_{\text{BF}}}{Z_{\text{BF}}}, \tag{4.55}$$

where $Z(\boldsymbol{T}_i)$ is the cost of route i in solution $\boldsymbol{T}$, and Z_{BF} is the cost of the best solution found for a problem instance during all computational tests on constructive heuristics, except for *bccm-IF* instances, where Z_{BF} was taken from the best metaheuristic solutions reported in [68].

For vehicle fleet size evaluation the fleet size excess, K_{gap}, for a heuristic solution was calculated as:

$$K_{\text{gap}} = |\boldsymbol{T}| - K_{\text{BF}}, \tag{4.56}$$

where K_{BF} is the minimum fleet size found for a problem instance during all computational tests on constructive heuristics. The required number of vehicles of solutions are not reported by Polacek et al. [68] since the authors solve an unlimited fleet size version of the CARPTIF. We refer to the Z_{BF} and K_{BF} values of random heuristics as the random variables $\tilde{Z}_{\text{gap}}$ and $\tilde{K}_{\text{gap}}$. During our computational tests, heuristics were evaluated over two objectives. One was their ability to find minimum cost solutions and the other to find minimum fleet size solutions. We refer to these objectives as min Z and min K. For each heuristic tested, the best solution for each objective is returned by Algorithm 4.1 .

To compare the deterministic and randomised versions of a heuristic we first calculated the expected number of runs required by a randomised version to produce a better solution than its deterministic version. A randomised heuristic performs a sequence of runs and since each run is independent, the runs can be modelled as a series of Bernoulli trials. A success is then a run that produces a solution that is equal to or better than the deterministic version's best solution. Let $Z^{\text{DT}}_{\text{gap}}$ and $K^{\text{DT}}_{\text{gap}}$ be the gaps of the best deterministic solutions for min Z and min K. If p is the probability of a success, which can be empirically calculated, the expected number of runs, α_e, to find the first success is $\frac{1}{p}$. To calculate α_e, each randomised heuristic was executed with 10 000 runs, and Z_{gap} and K_{gap} were calculated for each run before and after applying *Reduce-Vehicles*. The number of runs out of 10 000 that produced solutions with $Z_{\text{gap}} \leq Z^{\text{DT}}_{\text{gap}}$ was then used to calculate p; the same approach was used to calculate break-even points on $K^{\text{DT}}_{\text{gap}}$.

As an illustration on the *Cen-IF-b* problem instance, Figure 4.2 shows a histogram of Z_{gap} values for 10 000 runs of RM and the one run of IM. On the *Cen-IF-b* problem instance IM had gap values of $Z_{\text{gap}} = 0.5\%$ and $K_{\text{gap}} = 9$. Out of the 10 000 RM runs

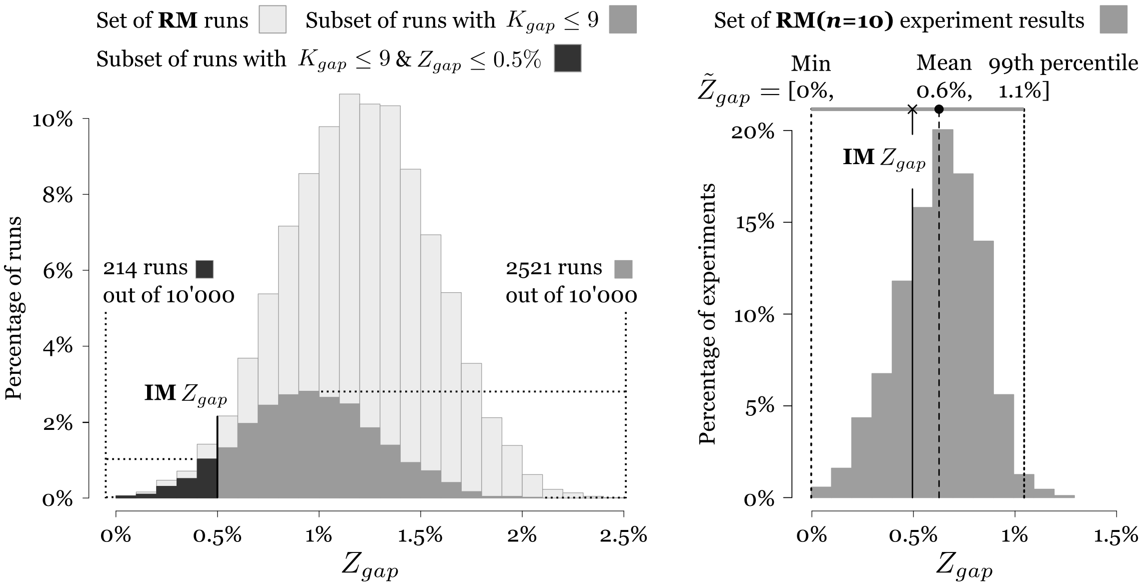


Figure 4.2: Histogram of cost gaps for *Improved-Merge* (IM) and *Randomised-Merge* (RM) on the *Cen-IF-b* problem instance.

251 had a solution with $Z_{\text{gap}} \leq 0.5\%$. By setting $p = 0.0251$, the expected number of runs required by RM to break-even with IM is thus $\alpha_e = 40$ when minimising Z is the only objective. Out of the 10 000 RM runs, 2521 (as shown in the figure) had a solution with $K_{\text{gap}} \leq 9$ giving RM a break-even point of $\alpha_e = 4$ when minimising K is the only objective. Lastly, 214 runs (also shown in the figure) gave a solution with $Z_{\text{gap}} \leq 0.5\%$ and $K_{\text{gap}} \leq 9$, giving RM a break-even point of $\alpha_e = 47$ over both objectives. Break-even points depend on the primary objectives. Referring to our example, if minimising Z is the primary objective, p is calculated using the number of runs where $Z_{\text{gap}} < Z_{\text{gap}}^{DT}$ or $Z_{\text{gap}} = Z_{\text{gap}}^{\text{DT}}$ and $K_{\text{gap}} < K_{\text{gap}}^{\text{DT}}$. Similarly if minimising K is the primary objective, p is calculated using the number of runs where $K_{\text{gap}} < K_{\text{gap}}^{\text{DT}}$ or $K_{\text{gap}} = K_{\text{gap}}^{\text{DT}}$ and $Z_{\text{gap}} < Z_{\text{gap}}^{\text{DT}}$.

A concise way of comparing RM and IM is through a statistical $\tilde{Z}_{\text{gap}}$ interval, which starts at the min Z_{gap} solution found over the 10 000 runs, and ends at the calculated 99th percentile value. Also included in the interval is the calculated mean of $\tilde{Z}_{\text{gap}}$; the interval for $\tilde{K}_{\text{gap}}$ is similarly defined. As an illustration, Figure 4.2 further shows a histogram of observed Z_{gap} values for RM($\alpha = 10$) over 1000 experiments[3]. The observed $\tilde{Z}_{\text{gap}}$ interval for RM($\alpha = 10$) on *Cen-IF-b* is $[0\%, 0.6\%, 1.1\%]$. The mean of RM($\alpha = 10$) is higher than IM's Z_{gap} since 40 runs is required for RM to break even with IM. The $\tilde{Z}_{\text{gap}}$ interval can be also be directly calculated using the 10 000 runs, eliminating the need to empirically calculate the interval for different α levels. The interval can be calculated using the probability mass function of a binomial distribution, given in Equations (4.57) and (4.58):

$$P(s) = \binom{\alpha}{s} p^s (1-p)^{\alpha - s}, \tag{4.57}$$

$$P(s \geq 1) = 1 - P(s = 0). \tag{4.58}$$

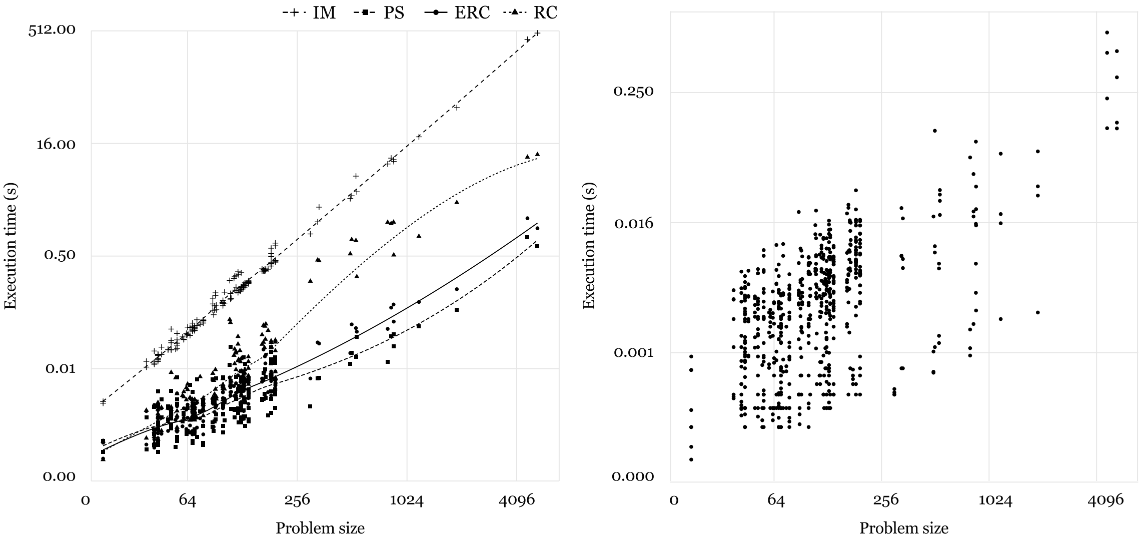
Here s is the number of required successes, α is the number runs, and p the probability of a success. The probability of a success is the probability that the heuristic will return a solution in a single run with Z_{gap} or K_{gap} below a certain threshold. Referring to Figure 4.2, the probability that RM will return a solution with $Z_{\text{gap}} \leq 0.5\%$ and $K_{\text{gap}} \leq 9$ is $\frac{214}{10000}$. Using Equations (4.57) and (4.58) we calculate that RM($\alpha = 10$) will find a better solution than IM on one or more of its 10 runs with a probability of 19%. By using this method in conjunction with RM run observations, the interval of RM can be calculated at specific α values. For all our computational tests in this chapter the interval lower limit is taken as the min Z_{gap} solution value over all the runs, and the 99th percentile upper limit is calculated via Equations (4.57) and (4.58) as the point where $P(k \geq 1) \approx 99\%$. The interval mean, $\overline{Z}_{\text{gap}}$, is calculated as the point where $P(k \geq 1) \approx \frac{1}{\alpha}$. When min K is the primary objective, the same interval can be calculated for K_{gap}, before and after applying *Reduce-Vehicles*. We refer to the $\tilde{Z}_{\text{gap}}$ interval as $[Z_{\text{gap}}^{\text{min}}, \overline{Z}_{\text{gap}}, Z_{\text{gap}}^{99^{\text{th}}}]$ and to the $\tilde{K}_{\text{gap}}$ interval as $[K_{\text{gap}}^{\text{min}}, \overline{K}_{\text{gap}}, K_{\text{gap}}^{99^{\text{th}}}]$. For *Cen-IF-b*, using observations from the 10000 RM runs the interval of $\tilde{Z}_{\text{gap}}$ for RM($\alpha = 10$) was calculated as $[0\%, 0.7\%, 1.1\%]$, which gives a slightly higher $\overline{Z}_{\text{gap}}$ than the empirically observed interval shown in Figure 4.2. In limited comparisons between the observed and calculated mean we found the calculated mean to always be slightly higher, but by no more than 0.5%. For all randomised heuristics we captured the Z_{gap} and K_{gap} values over 10 000 runs and used the results to calculate break-even points and intervals for the min K and min Z objectives. Intervals were calculated for both the primary and secondary objectives.

[3] The problem instance was solved 1000 times using RM($\alpha = 10$). For each experiment the Z_{gap} value of the best solution returned over the 10 runs was captured.

4.4.2 Computational time and break even analysis

The CPU time required by heuristics to produce a single solution, which is the same for the deterministic and randomised versions, plus the CPU time taken by *Reduce-Vehicles* on a solution are shown in Figure 4.3a, and averages of benchmark sets are shown in Table 4.1.

The CPU time of *Reduce-Vehicles* per solution is constant for all the heuristics and is shown in Figure 4.3b. The CPU time of IM includes its initialisation phase, which took between three and four times longer than its execution phase. CPU times of ERC and RC to produce a single solution also include the CPU time to construct an initial giant tour. On the two largest *Cen-IF* instances, IM took eight minutes to produce a single feasible solution, whereas PS took less than a second. ERC was also efficient, capable of solving large *Cen-IF* instances in 1.5 seconds. Results further show that *Reduce-Vehicles* is efficient, taking at most one second on large *Cen-IF* instances. The CPU times of



(a) CPU time of constructive heuristics.

(b) CPU time of *Reduce-Vehicles*.

Figure 4.3: Problem instance size ($|\boldsymbol{R}|$) versus CPU time (in seconds), on log-xy axis, of deterministic heuristics to produce one solution and for *Reduce-Vehicles* to improve the solution.

Table 4.1: Average heuristic CPU times (in seconds) to generate one solution per benchmark set.

Set	IM	PS	ERC	RC	*Reduce-Vehicles*
Cen-IF	306.71	0.55	0.99	8.16	0.37
Act-IF	4.14	0.01	0.03	0.73	0.01
Lpr-IF	4.39	0.02	0.04	0.48	0.01
mval-IF-3L	0.19	0.01	0.01	0.02	< 0.01
bccm-IF-3L	0.21	< 0.01	0.01	0.01	< 0.01
gdb-IF-3L	0.04	< 0.01	< 0.01	< 0.01	< 0.01
bccm-IF	0.24	0.01	0.01	0.01	0.01
gdb-IF	0.05	0.01	< 0.01	0.01	< 0.01
Global mean	39.50	0.10	0.187	1.35	0.10

IM: *Improved-Merge*; PS: *Path-Scanning*; ERC: *Efficient-Route-Cluster*; RC: *Route-Cluster.*

RC were on average under one second on all sets except *Cen-IF* where RC was under ten seconds. On small instances with $|\boldsymbol{R}| \leq 256$, all randomised heuristics can complete 100 runs in less than ten seconds, including RM whose most expensive component is its initialisation phase. The high computational times of IM is only a factor on realistically sized instances with $|\boldsymbol{R}| > 256$. The tested versions of IM and RM are too slow for near real-time decision support, but additional effort can be invested to improve their implementation code efficiency, depending on how well they perform in minimising Z and K compared to the already efficient heuristics.

To compare the deterministic heuristics with their randomised versions, the number of runs, α_e, required by the randomised versions to break even was calculated for the min Z_{gap} and min K_{gap} objectives. In case of the latter, *Reduce-Vehicles* was applied to both the deterministic and random versions. The number of runs per deterministic heuristic are one run for IM, eight runs for PS, and six runs for ERC and RC. Average break-even values for PS-RL against PS, RM against IM, ERC-RL against ERC and RC-RL against RC per benchmark set are shown in Table 4.2. Averages were calculated by excluding problem instances for which randomised heuristics failed to find a better solution in 10000 runs. The number of instances per set for which this was the case is shown in Table 4.3. On most instances the randomised heuristics are capable of outperforming their deterministic versions if allowed sufficient number of runs. Comparing α_e against the number of runs required by the deterministic versions, α_e was always equal or higher. When min Z was the primary objective, ERC-RL and RC-RL required the least number of runs to break-even and RM required the most. PS-RL broke even on all but 1 of the 159 instances, and its break-even point was on average under 100 runs. When min K was the primary objective, all the heuristics required more runs to break-even, indicating that the deterministic rules result in solutions requiring fewer vehicles. On small problem instances, computation time is not a factor and randomised heuristics can be allowed high-run levels to outperform deterministic versions. The break-even points were also lower on these instances. This was not the case on more realistic instances. Randomised heuristics required more runs to break-even which, in turn, also require more computation time. Contrary to tests on small instances, the choice between randomised and deterministic versions is not clear for realistic instances and will depend on available computational time and the solution quality of heuristics. These factors are analysed next.

Table 4.2: Average number of runs, α_e, required by randomised heuristics to break-even with their deterministic versions.

	min Z primary objective				min K primary objective			
Set	RM	PS-RL	ERC-RL	RC-RL	RM	PS-RL	ERC-RL	RC-RL
Cen-IF	14	48	-	-	5	48	502	-
Act-IF	13	24	115	24	69	24	115	24
Lpr-IF	1024	111	24	16	946	117	31	48
mval-IF-3L	8	58	10	11	7	53	11	13
bccm-IF-3L	7	114	12	8	4	111	15	47
gdb-IF-3L	41	35	7	6	23	178	6	6
bccm-IF	11	68	32	16	21	73	45	21
gdb-IF	368	78	51	27	908	52	60	49
Global mean	186	67	36	15	248	82	98	30

IM and RM: *Improved* and *Randomised-Merge*; PS: *Path-Scanning*; ERC: *Efficient-Route-Cluster*; RC: *Route-Cluster*; RL: *Random-Link* multi-start versions.

Table 4.3: Number of problem instances on which randomised heuristics failed to break-even over 10000 runs.

	# instances	min Z_{gap} primary objective				min K_{gap} primary objective			
Set	in set	RM	PS-RL	ERC-RL	RC-RL	RM	PS-RL	ERC-RL	RC-RL
Cen-If	3	0	0	3	3	0	0	2	3
Act-If	3	2	0	0	0	2	0	0	0
Lpr-IF	15	5	1	0	0	2	1	1	0
mval-IF-3L	34	0	0	0	0	0	0	0	0
bccm-IF-3L	34	0	0	0	0	0	0	0	0
gdb-IF-3L	23	0	0	0	0	0	0	0	0
bccm-IF	34	0	0	0	0	0	0	0	1
gdb-IF	23	4	0	0	0	0	0	3	1
Total	169	11	1	3	3	4	1	6	5

IM and RM: *Improved* and *Randomised-Merge*; PS: *Path-Scanning*; ERC: *Efficient-Route-Cluster*; RC: *Route-Cluster*; RL: *Random-Link* multi-start versions.

4.4.3 Performance evaluation

The aim of our computational tests was to identify the best performing heuristic, subject to computation time-limits encountered in practical applications. The second aim was to determine if heuristic performance was different between the primary objectives of minimising Z or K, and if it differed between different benchmark sets, particularly between more realistic sets and those proposed in literature. Based on the computational times and break-even points of heuristics (Tables 4.1 and 4.2), the randomised heuristics were evaluated as follows. On small *gdb* and *bccm* related benchmark sets as well as *mval-IF-3L*, the Z_{gap} and K_{gap} primary and secondary objective intervals were calculated at $\alpha = 1000$, and on *Lpr-IF*, *Act-IF* and *Cen-IF* at $\alpha = 100$. Computational times per problem instance for these setups are shown in Figure 4.4. With $\alpha = 1000$, the execution times of PS-RL and ERC-RL fell below one minute on small instances, and with $\alpha = 100$, both heuristics have total computation times of less than two minutes on the largest *Cen-IF* instances, except for one problem instance where ERC-RL required three minutes, which we assume to be acceptable for near real-time decision support. The same criteria would disqualify IM and RM from implementation, and limit RC-RL to less than ten runs; but as mentioned, their implementation code efficiency can be improved. ERC-RL and RC-RL were compared in Chapter 3 in which we showed that RC-RL performs marginally better than ERC-RL under the same execution time-limits. As mentioned, we subsequently favoured ERC-RL since it is easier to implement. Accordingly, we only report on aggregated results of our tests on ERC-RL.

To measure the impact of *Reduce-Vehicles*, the fleet size before and after its application was measured on deterministic heuristic solutions. The $\tilde{K}_{\text{gap}}$ interval calculations were also completed without and with its application and the resulting intervals compared. Table 4.4 shows the number of problem instances per set on which *Reduce-Vehicles* decreased the number of required vehicles for deterministic solutions, and on which it decreased any of the three interval values on randomised heuristics. Despite its simplicity, the application of *Reduce-Vehicles* improved the performance of heuristics, and because of its computational efficiency (Figure 4.3b) it did so without significantly increasing the computation times. It can thus be applied to any heuristics when real-time decision support is needed and min K is an objective, or if the fleet sizes of solutions exceed the fleet limit. In the remainder

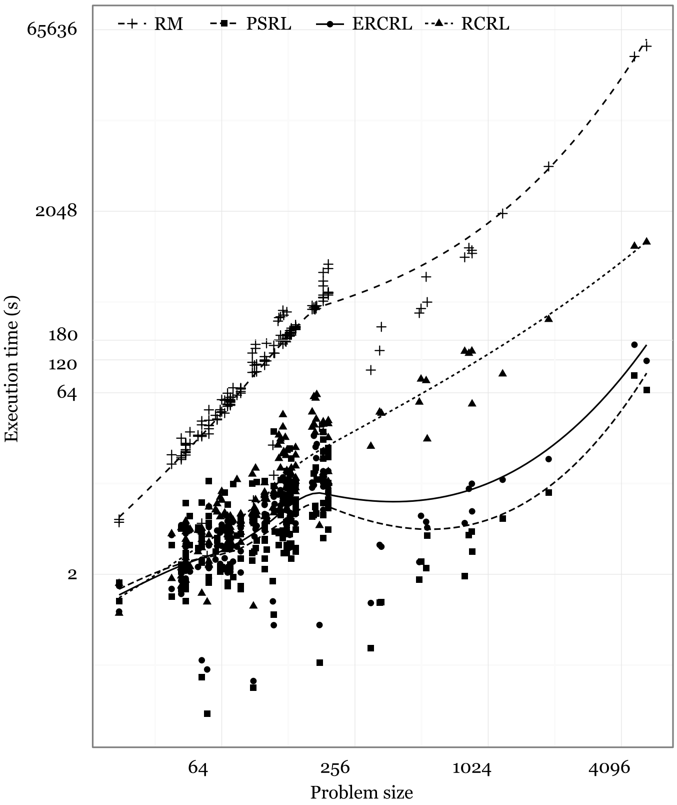


Figure 4.4: Problem instance size ($|\boldsymbol{R}|$) versus execution time of *Randomised-Merge* (RM), *Path-Scanning-Random-Link* (PSRL), *Efficient-Route-Cluster-Random-Link* (ERCRL) and *Route-Cluster-Random-Link* implementations.

Table 4.4: Number of problem instances on which *Reduce-Vehicles* was able to reduce the fleet size.

Set	# instances	IM	RM	PS	PS-RL	ERC	ERC-RL	RC	RC-RL
Cen-IF	3	2	2	2	2	3	2	3	1
Act-IF	3	2	1	0	0	0	0	0	0
Lpr-IF	15	10	9	2	2	11	2	6	1
mval-IF-3L	34	29	21	13	11	22	17	19	15
bccm-IF-3L	34	30	17	3	3	16	7	11	4
gdb-IF-3L	23	20	1	3	2	15	5	8	1
bccm-IF	34	34	26	32	31	34	34	34	34
gdb-IF	23	19	9	14	9	22	21	22	20
Total	169	146	86	69	60	123	88	103	76
Fraction		0.86	0.51	0.41	0.36	0.73	0.52	0.61	0.45

IM and RM: *Improved* and *Randomised-Merge*; PS: *Path-Scanning*; ERC: *Efficient-Route-Cluster*; RC: *Route-Cluster*; RL: *Random-Link* multi-start versions.

of this section, results reported on heuristic costs and fleet sizes are with the application of *Reduce-Vehicles* when min K is the primary objective, and without it when min Z is the primary objective.

Results for deterministic and randomised heuristics on the waste collection benchmark sets, *Lpr-IF*, *Act-IF* and *Cen-IF*, are shown in Figure 4.5 and includes Z_{gap} values when min Z was the primary objective, and K_{gap} values when min K was the primary objective. On these sets, the best performing heuristic differed per set and per primary objective.

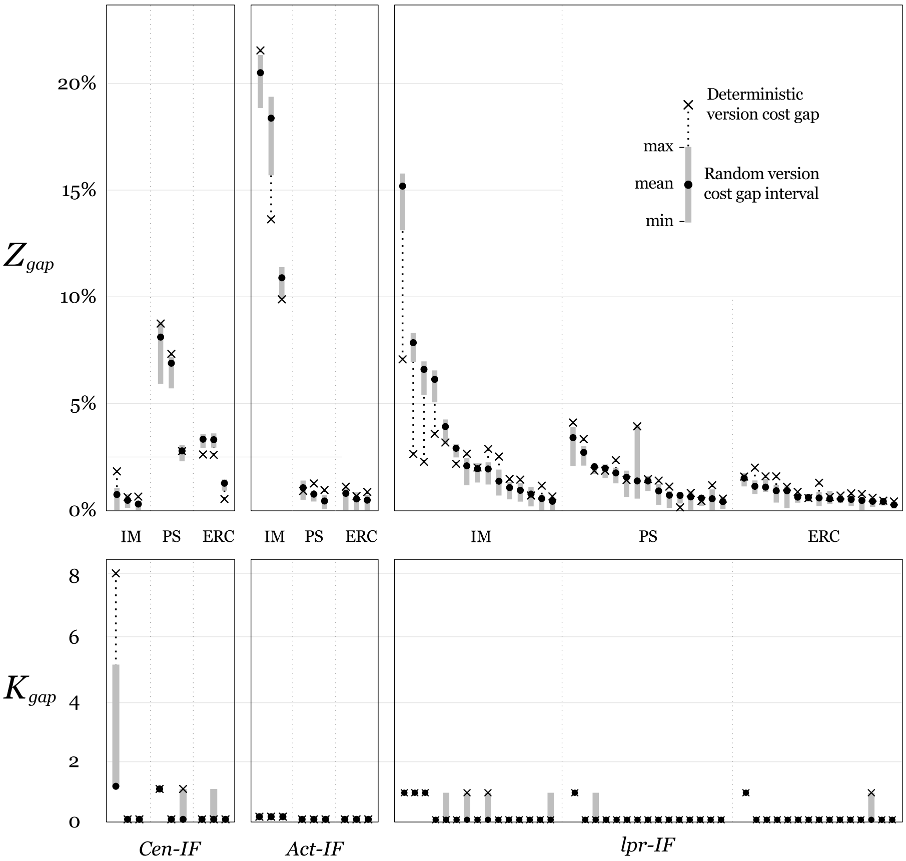


Figure 4.5: Results on waste collection benchmark instances of the deterministic and randomised versions of *Merge* (M), *Path-Scanning* (PS) and *Efficient-Route-Cluster* (ERC) when min Z or min K is the primary objective.

RM and IM performed the best on *Cen-IF* in minimising Z but they performed poorly in minimising K with IM exceeding the minimum fleet size solution by eight vehicles on *Cen-IF-b*. RM had a $\tilde{K}$ interval of [1.0, 1.0, 5.0] for the same problem instance. IM and RM also performed the worst for *Lpr-IF* in minimising K. ERC-RL performed the best for *Act-IF* and *Lpr-IF* whereas ERC (the deterministic version) performed better for all three *Cen-IF* instances. Both versions performed the best for all three instances in minimising K, followed closely by PS and PS-RL. Despite the application of *Reduce-Vehicles*, all heuristics failed to find min fleet size solutions for a few instances. ERC-RL was the most consistent of the randomised heuristics, with small $\tilde{Z}$ intervals spanning less than 1%. RM had the largest intervals in excess of 5% for some of the problem instances. The intervals on $\tilde{K}$ were small for all heuristics, except RM and IM for *Cen-IF-b*. ERC and ERC-RL were always within 5% of the minimum Z solution found, and PS and PS-RL within 10% on *Cen-IF* and within 5% on *Lpr-IF*. IM and RM were also within 10% on most instances, except one *Lpr-IF* problem instance and the *Act-IF* set where it performed very poorly compared to other heuristics.

Summary Z_{gap} results for the heuristics on all the benchmark sets are shown in Table 4.5. Full results for *Cen-IF*, *Act-IF* and *Lpr-IF* can be found in [87], as well as Appendix A.1.1, including computational times for the different setups. Full results for the smaller benchmark sets can be also be found in [87], as well as Appendix A.1.2.

Results are shown when min Z is either the primary or secondary objective, and for the five smaller benchmark sets the randomised heuristics were analysed at $\alpha = 10\,000$. Results for the smaller sets further highlight the impact of benchmark sets on heuristic performance. The increase in α resulted in the random heuristics dominating their deterministic counterparts, whereas the performance of the deterministic counterparts were much closer for the waste collection sets, and even better in a few cases. In addition to *Cen-IF*, RM performed the best for *bccm-IF* and *gdb-IF*, but it performed the worst for all other sets. Routes for *bccm-IF* and *gbd-IF* typically consist of a single subtrip, indicating that RM and IM performs better when this is the case, and possibly when the depot coincides with an IF, as is the case for *Cen-IF*. ERC-RL and RC-RL struggled with *bccm-IF* and *gbd-IF*, producing solutions with $\overline{Z}_{\text{gap}}$ values that were on average 48% and 20% from Z_{BF}, respectively. This implies that the expected costs, $\overline{Z}$, of the heuristics were on average 1.5 and 1.2 times more than Z_{BF}. These were the largest gaps observed during all the computational tests. On all the other sets except *Cen-IF*, ERC-RL performed the best. As expected, RC and RC-RL performed better than ERC and ERC-RL, more so on smaller sets. For time critical applications, the more efficient and easier to implement ERC and ERC-RL heuristics can be used instead of RC and RC-RL as their solutions will still be better than those of other heuristics. For all the benchmarks, PS and PS-RL were the most consistent with the best global Z_{gap} mean, despite not being the best performing heuristic on any specific set. The $\tilde{Z}_{\text{gap}}$ intervals of PS-RL were however the largest, whereas ERC-RL and RC-RL had the smallest intervals, even for benchmarks where they performed poorly. When min Z was the secondary objective, Z_{gap} values were higher in a few cases as expected; but for a number of heuristics, particularly on smaller sets, the values were lower. This can be attributed to *Reduce-Vehicles* which was able to reduce total costs through its route removal and arc insertion procedures. The cost reduction was most prominent for deterministic solutions and for $Z_{\text{gap}}^{\text{min}}$ values. Treating min Z as a secondary objective did not have a significant impact on the solution costs of heuristics.

Summary K_{gap} results for heuristics on all benchmark sets together with min K as primary and secondary objectives are shown in Table 4.6. The intervals were calculated at $\alpha = 1000$ for randomised heuristics for small benchmark sets, and at $\alpha = 100$ for *Cen-IF*, *Act-IF* and *Lpr-IF*. Similar to Z_{gap} analysis, results were different between waste collection and small benchmark sets, and between heuristics. All heuristics were able to find min fleet solutions for *Act-IF*. On the other waste collection sets, RC-RL performed the best followed by RC, ERC-RL, PS-RL, ERC and PS. IM and RM performed the worst. On small *bccm-IF* and *gdb-IF* benchmarks ERC and RC again struggled, exceeding the minimum fleet size on average by 3.5 and 7 vehicles. Both performed better than RM and IM on *mval-IF-3L* and *bccm-IF-3L*, and all the randomised heuristics performed similarly for *gdb-IF-3L*. PS and PS-RL performed the best on small benchmark sets and were again the most consistent over all sets with the best global Z_{gap} means. In all the cases the randomised heuristics performed better than their deterministic versions, and their intervals were also small. When min K was treated as a secondary instead of primary objective, the increase in K_{gap} values was in excess of one vehicle on ERC and ERC-RL, and close to one vehicle for IM and RM. PS and PS-RL results were similar when min K was the primary and secondary objective.

Table 4.5: Average Z_{gap} values (in %) on benchmark sets when min Z is the primary or secondary objective.

Min Z	Set	IM	RM	PS	PS-RL	ERC	ERC-RL	RC	RC-RL
1st Obj	*Cen-IF*	1.0	[0.0, 0.5, 0.7]	6.3	[4.7, 6.0, 6.4]	1.9	[2.3, 2.6, 2.9]	1.7	[2.1, 2.5, 2.7]
	Act-IF	15.0	[14.8, 16.6, 17.4]	0.9	[0.0, 0.6, 0.9]	1.0	[0.3, 0.7, 1.0]	0.8	[0.1, 0.5, 0.7]
	Lpr-IF	2.4	[2.8, 3.6, 3.9]	1.7	[0.8, 1.4, 1.8]	1.0	[0.4, 0.7, 0.9]	0.7	[0.1, 0.4, 0.6]
	Mean	6.1	[5.9, 6.9, 7.3]	3.0	[1.8, 2.7, 3.0]	1.3	[1.0, 1.3, 1.6]	1.1	[0.8, 1.1, 1.3]
	mval-IF-3L	18.0	[8.2, 12.4, 14.6]	13.6	[3.6, 6.2, 9.3]	14.7	[8.0, 8.3, 10.7]	13.0	[6.3, 6.4, 8.4]
	bccm-IF-3L	31.5	[17.1, 17.3, 20.1]	10.1	[1.6, 4.0, 6.4]	14.0	[5.9, 6.1, 8.9]	10.8	[3.9, 4.0, 5.5]
	gdb-IF-3L	11.7	[1.5, 3.0, 4.2]	7.8	[1.2, 2.6, 4.7]	11.3	[4.1, 4.5, 5.9]	8.8	[2.2, 2.5, 3.7]
	bccm-IF	18.1	[7.5, 9.9, 13.0]	25.4	[8.5, 12.3, 16.7]	68.6	[47.6, 48.0, 52.6]	68.5	[47.4, 47.7, 52.2]
	gdb-IF	11.0	[1.4, 2.2, 3.1]	11.2	[1.7, 3.4, 5.2]	34.9	[20.0, 20.2, 24.2]	34.5	[19.8, 20.0, 24.0]
	Mean	18.1	[7.1, 9.0, 11.0]	13.6	[3.3, 5.7, 8.5]	28.7	[17.1, 17.4, 20.5]	27.1	[15.9, 16.1, 18.8]
2nd Obj	*Cen-IF*	2.4	[1.1, 1.3, 2.3]	6.3	[4.7, 5.9, 6.4]	3.0	[3.0, 4.0, 4.5]	2.5	[2.5, 3.1, 3.6]
	Act-IF	10.3	[10.7, 15.7, 16.0]	0.9	[0.0, 0.6, 0.9]	1.0	[0.3, 0.7, 1.0]	0.8	[0.1, 0.5, 0.7]
	Lpr-IF	2.9	[2.6, 3.5, 3.9]	1.5	[0.8, 1.4, 1.8]	1.4	[0.7, 1.1, 1.3]	0.9	[0.2, 0.5, 0.8]
	Mean	5.2	[4.8, 6.8, 7.4]	2.9	[1.8, 2.6, 3.0]	1.8	[1.3, 1.9, 2.3]	1.4	[0.9, 1.4, 1.7]
	mval-IF-3L	16.3	[2.6, 9.1, 10.0]	12.6	[3.0, 5.7, 7.6]	15.2	[4.5, 8.3, 9.0]	13.5	[2.6, 6.4, 7.1]
	bccm-IF-3L	28.4	[8.4, 17.3, 18.0]	9.9	[1.6, 4.0, 6.4]	14.8	[4.1, 6.2, 7.7]	11.2	[1.3, 4.0, 4.6]
	gdb-IF-3L	13.3	[0.4, 2.3, 3.4]	7.8	[1.2, 2.5, 4.7]	12.6	[1.7, 4.4, 5.1]	9.6	[1.4, 2.3, 3.2]
	bccm-IF	15.5	[5.6, 8.6, 9.9]	21.4	[7.8, 11.4, 15.1]	69.0	[41.4, 48.0, 48.6]	68.9	[41.0, 47.7, 48.4]
	gdb-IF	10.4	[0.9, 1.9, 2.8]	9.3	[1.3, 2.9, 4.7]	35.5	[15.6, 20.2, 20.7]	34.6	[15.5, 20.0, 20.5]
	Mean	16.8	[3.6, 7.8, 8.8]	12.2	[3.0, 5.3, 7.7]	29.4	[13.5, 17.4, 18.2]	27.6	[12.4, 16.1, 16.8]
1st Obj	*Global*	13.6	[6.7, 8.0, 9.4]	9.6	[2.8, 4.4, 6.2]	18.4	[11.1, 11.3, 13.2]	17.4	[10.2, 10.4, 12.1]
2nd Obj	*Global*	12.4	[4.0, 7.4, 8.0]	8.7	[2.6, 4.1, 5.7]	19.1	[8.9, 11.4, 12.1]	17.8	[8.1, 10.4, 11.0]

$[Z_{\text{gap}}^{\min}, \overline{Z}_{\text{gap}}, Z_{\text{gap}}^{99^{\text{th}}}]$: Results interval for random heuristics; *Mean*: Average over set averages. *Global*: Average over all set averages. 1st Obj: Min Z was the primary objective and min K secondary; 2nd Obj: Min K was the primary objective and min Z secondary; IM and RM: *Improved* and *Randomised-Merge*; PS: *Path-Scanning*; ERC: *Efficient-Route-Cluster*; RC: *Route-Cluster*; RL: *Random-Link* multi-start versions.

Table 4.6: Average K_{gap} values on benchmark sets when min K is the primary or secondary objective.

Min K	Set	IM	RM	PS	PS-RL	ERC	ERC-RL	RC	RC-RL
1st Obj	*Cen-IF*	2.7	[0.3, 1.3, 1.7]	0.7	[0.3, 0.7, 0.7]	0.7	[0.0, 0.3, 0.3]	0.3	[0.0, 0.0, 0.0]
	Act-IF	0.0	[0.0, 0.0, 0.0]	0.0	[0.0, 0.0, 0.0]	0.0	[0.0, 0.0, 0.0]	0.0	[0.0, 0.0, 0.0]
	Lpr-IF	0.3	[0.2, 0.4, 0.5]	0.1	[0.1, 0.1, 0.1]	0.2	[0.1, 0.1, 0.1]	0.1	[0.1, 0.1, 0.1]
	Mean	1.0	[0.2, 0.6, 0.7]	0.3	[0.1, 0.3, 0.3]	0.3	[0.0, 0.1, 0.1]	0.1	[0.0, 0.0, 0.0]
	mval-IF-3L	0.7	[0.2, 0.2, 0.3]	0.3	[0.1, 0.1, 0.2]	0.6	[0.2, 0.2, 0.3]	0.6	[0.1, 0.1, 0.2]
	bccm-IF-3L	1.2	[0.2, 0.2, 0.3]	0.1	[0.0, 0.0, 0.1]	0.4	[0.0, 0.0, 0.1]	0.3	[0.0, 0.0, 0.0]
	gdb-IF-3L	0.7	[0.0, 0.0, 0.0]	0.1	[0.0, 0.0, 0.0]	0.5	[0.0, 0.0, 0.0]	0.3	[0.0, 0.0, 0.0]
	bccm-IF	1.1	[0.1, 0.5, 0.6]	1.2	[0.1, 0.4, 0.7]	6.7	[3.9, 3.9, 4.2]	6.7	[3.9, 3.9, 4.2]
	gdb-IF	0.9	[0.1, 0.2, 0.3]	0.6	[0.0, 0.1, 0.3]	3.4	[1.3, 1.3, 1.7]	3.4	[1.3, 1.3, 1.7]
	Mean	0.9	[0.1, 0.2, 0.3]	0.5	[0.0, 0.1, 0.3]	2.3	[1.1, 1.1, 1.3]	2.3	[1.1, 1.1, 1.2]
2nd Obj	*Cen-IF*	3.3	[3.7, 5.0, 4.0]	0.7	[0.3, 0.7, 0.7]	7.0	[8.3, 9.3, 9.3]	3.7	[4.0, 4.7, 5.3]
	Act-IF	0.7	[0.3, 0.3, 0.3]	0.0	[0.0, 0.0, 0.0]	0.0	[0.0, 0.0, 0.0]	0.0	[0.0, 0.0, 0.0]
	Lpr-IF	0.9	[0.9, 1.1, 1.1]	0.1	[0.1, 0.1, 0.1]	1.1	[0.9, 1.2, 1.3]	0.9	[0.7, 0.8, 0.9]
	Mean	1.6	[1.6, 2.1, 1.8]	0.3	[0.1, 0.3, 0.3]	2.7	[3.1, 3.5, 3.5]	1.5	[1.6, 1.8, 2.1]
	mval-IF-3L	1.3	[0.7, 0.9, 1.1]	0.4	[0.1, 0.2, 0.3]	1.1	[0.6, 0.6, 0.9]	0.7	[0.4, 0.4, 0.6]
	bccm-IF-3L	2.2	[0.6, 0.6, 0.9]	0.1	[0.0, 0.0, 0.1]	0.6	[0.2, 0.2, 0.5]	0.4	[0.1, 0.1, 0.2]
	gdb-IF-3L	1.9	[0.0, 0.3, 0.4]	0.1	[0.0, 0.0, 0.1]	0.8	[0.4, 0.3, 0.7]	0.5	[0.2, 0.1, 0.3]
	bccm-IF	1.9	[0.4, 0.7, 1.2]	1.6	[0.2, 0.5, 0.9]	6.9	[4.6, 4.6, 5.2]	6.9	[4.6, 4.6, 5.1]
	gdb-IF	1.2	[0.3, 0.3, 0.5]	0.8	[0.1, 0.2, 0.4]	3.6	[1.9, 1.9, 2.5]	3.5	[1.9, 1.9, 2.5]
	Mean	1.7	[0.4, 0.6, 0.8]	0.6	[0.1, 0.2, 0.4]	2.6	[1.5, 1.5, 2.0]	2.4	[1.4, 1.4, 1.7]
1st Obj	*Global*	1.0	[0.1, 0.3, 0.4]	0.4	[0.1, 0.1, 0.2]	1.6	[0.7, 0.7, 0.8]	1.5	[0.7, 0.7, 0.8]
2nd Obj	*Global*	1.7	[0.9, 0.9, 1.2]	0.5	[0.1, 0.2, 0.3]	2.6	[2.1, 2.1, 2.6]	2.1	[1.5, 1.6, 1.8]

$[K_{\text{gap}}^{\text{min}}, \overline{K}_{\text{gap}}, K_{\text{gap}}^{99\text{th}}]$: Results interval for random heuristics; *Mean*: Average over set averages. *Global*: Average over all set averages. 1st Obj: Min K was the primary objective and min Z secondary; 2nd Obj: Min Z was the primary objective and min K secondary; IM and RM: *Improved* and *Randomised-Merge*; PS: *Path-Scanning*; ERC: *Efficient-Route-Cluster*; RC: *Route-Cluster*; RL: *Random-Link* multi-start versions.

Compared to other heuristics, PS and PS-RL were expected to do well in meeting fleet limits. During their construction process arcs are added to a route until it reaches its limit. As subtrips become full, low demand arcs can still be included in subtrips; these are added by PS regardless of how far they are from the current route position. The construct mechanisms of IM and RM result in them struggling with fleet limits. Routes progressively became longer as more mergers took place; however, the mergers were made only according to cost savings. Directly merging two routes results in better savings than merging them through an IF visit. IM and RM thus have a tendency to produce long routes that cannot be merged through an IF visit without exceeding the route duration limit, when one route can be easily split and incorporated into the other routes. *Reduce-Vehicles* is effective in reducing the fleet size in these situations. ERC, RC, ERC-RL and RC-RL take giant routes as their input, which are constructed without taking arc demand or service time into consideration. If a giant route consists of successive high demand arcs, the heuristics will be forced to partition the route between these arcs, and the first portion will not be close to its demand limit. The same also occurs in high service time arc-sequences. *Reduce-Vehicles* then attempts to reinsert small demand and service time arcs in these routes, which is why the heuristics benefit from its application.

Results show that the choice of benchmark sets influence heuristic performance, as does the treatment of K. If tests were limited to the *gdb-IF* and *bccm-IF* benchmark sets proposed in literature, and the only objective is to minimise Z, RM would have performed the best, followed by PS-RL. ERC-RL and RC-RL would have been discarded based on their very poor performance on the sets. However, as shown in our benchmark set analysis in Section 2.4.2 these problem instances have features that are significantly different from waste collection instances. The assumption of an unlimited fleet size also rarely holds in practice. RC-RL and ERC-RL were the best performing heuristics in terms of minimising K and Z. For computational tests to be of greater practical value, results on realistic instances should be prioritised. The limited tests performed in this thesis indicate that ERC-RL linked with *Reduce-Vehicles* is best suited for application purposes where near real-time decision support is required and the primary objective is to either minimise Z or K. The deterministic version, ERC, can be used when available computational time is very limited. If small instances are to be solved, PS-RL is best suited for application purposes. Similarly, if instances are to be solved with unique characteristics, inconsistent with those of all benchmark sets solved in this paper, PS-RL would also be best suited as it is seems to be more robust for different types of instances.

A limitation of the analysis, and by extension the evaluations, is that tests were performed on only six realistic waste collection instances, three of each from the *Act-IF* and *Cen-IF* sets. Results on *Lpr-IF* may also be misleading since gap measurements from lower bounds or best solutions found are generally small for the benchmark set. Another limitation is that Z_{gap} and K_{gap} values were taken from the best solutions found during all our computational tests on heuristics. The gaps will thus be higher when taken from lower bounds and optimal solution values. The Z_{gaps} were high for *bccm-IF* instances, which were calculated from the metaheuristic solutions reported in [68]. The heuristics were able to match metaheuristic solutions for three out of 19 instances solved in [68], but on average the best performing heuristic, RM, had cost gaps of 7.5% when allowed 10 000 runs. Optimal solution gaps on other sets may also be in excess of 7.5% if one were to use *bccm-IF* as a baseline. The set did prove more difficult for the heuristics, with the best performing heuristic differing per problem instance. The same did not occur on realistic waste collection sets. Whether the poor performance of constructive heuristics on *bccm-IF* can be generalised to realistic waste collection instances remains an open question.

In terms of K_{gap} analysis, which is seldom presented in studies on CARPs, all heuristics had $K_{\text{gap}} > 0$ for some of the problem instances. Since K_{gap} values were taken from best solutions found by the heuristics, the gaps from the minimum fleet size will be equal or higher. If K was treated as being limited, instead of an objective, and the fleet limit was close to minimum, the heuristics would have failed to find feasible solutions on a number of instances, even with the application of *Reduce-Vehicles*.

4.5 Main findings

In this section, we evaluated constructive heuristics for the MCARPTIF. Constructive heuristics that can quickly generate feasible solutions for the MCARPTIF is an important area of research. In many practical applications such as the one that inspired our article on constructive heuristics [87], the problem has to be solved in near real-time. When faced with a new problem such as the MCARPTIF, which accurately models waste collection, constructive heuristics are used as a first step to solve the problem.

In contrast to its original formulation, studies on CARP heuristics treat the vehicle fleet as unlimited. In practical applications, the fleet is generally limited and our first research contribution of this chapter was the evaluation of MCARPTIF constructive heuristics on their ability to minimise the fleet size. To improve heuristic performance an efficient fleet size reduction procedure was developed and linked with the tested heuristics. This allowed heuristics to better deal with the objective, but despite its application, none of the heuristics could find minimum fleet solutions on all CARPTIF and MCARPTIF benchmark instances. More advanced procedures for minimising the number of required routes should yield better results, thus warranting further research on the topic, which we leave for future work.

Secondly, results showed that problem instance characteristics influence heuristic performance, but the lack of realistic waste collection benchmark sets makes this difficult to investigate. We have addressed this gap, admittedly to a limited extent, by the introduction of new realistic waste collection instances in Section 2.4.2. Results indicate that problem instance characteristics such as IF locations that are incident to vehicle depots and low route duration time-limits in relation to vehicle capacity, play a critical role in heuristic performance. Execution times were also high for the realistic problem instances, pointing to heuristic inefficiencies that would have gone unnoticed for smaller instances.

Lastly, the method that we employed to evaluate randomised heuristics is also of value to studies on CARPs. The heuristics were statistically analysed by modelling runs as a series of Bernoulli trials, and calculating total cost and vehicle fleet size intervals for specific run levels. An advantage of the analysis is that a single set of experiments can be used to calculate intervals for different run levels. The analyses were used to more accurately compare heuristic performance, and can be applied to any multi-start heuristic or metaheuristic where runs are independent.

4.6 Conclusion

An often cited advantage of CARP constructive heuristics is that they can be easily modified to extensions of the problem. The existing CARP and MCARP heuristics were indeed easy to modify for the MCARPTIF, but their performance on benchmark instances were inconsistent. In particular, the best performing MCARP heuristic, *Improved-Merge*, adapted to the MCARPTIF performed the best on two CARPTIF benchmarks proposed in literature, yet struggled with waste collection benchmark sets. The weakest MCARP heuristic, *Path-Scanning-Random-Link*, was the most robust over all benchmarks sets. On waste collection applications, *Efficient-Route-Cluster-Random-Link* linked with *Reduce-Vehicles* performed the best, yet it produced solutions with total costs 1.2 to 1.5 times more than that of other heuristics on the benchmarks currently proposed in literature. Given the lack of realistic MCARPTIF benchmark sets, identifying the best performing heuristic for waste collection applications proved difficult. Results indicate that of the tested heuristics, *Efficient-Route-Cluster* and *Efficient-Route-Cluster-Random-Link* are best suited when waste collection routing problems have to be solved within short executions times.

Results showed that although the constructive heuristics can quickly generate solutions, the quality of the solutions are, as expected, generally poor and inconsistent among the different heuristics. When more execution time is available, the solutions can thus be improved through Local Search (LS) and metaheuristics. In the next two chapters efficient LS improvement heuristics are developed. The developed LS methods are then used in Chapter 7 within a Tabu Search metaheuristic.

Chapter appendix

4.A Detailed algorithm descriptions

4.A.1 Multiple run solution constructor

Algorithm 4.1: *nConstruct*

```
Input  : number of solutions to be generated, $\alpha$, the Solution-Constructor, and fleet size limit,
         $K_{\text{UB}}$, if applicable.
Output: Min cost solution $\boldsymbol{T}^*_{minZ}$, min fleet size solution $\boldsymbol{T}^*_{minK}$, and min cost and limited fleet
         size solution $\boldsymbol{T}^*_{K_{\text{UB}}}$.
$Z^*_{minZ} = Z^*_{minK} = Z^*_{K_{\text{UB}}} = \infty$ // values for min cost solution //;
$K^*_{minZ} = K^*_{minK}$ // values for min fleet solution //;
$t = 1$;
while $t \leq \alpha$ do
    $\boldsymbol{T} =$ Solution-Constructor // solution constructed using a constructive heuristic //;
    $K = |\boldsymbol{T}|$ // number of vehicles used //;
    $Z = \sum_{j=1}^{K} Z(\boldsymbol{T}_j)$ // solution cost //;
    if $\left(Z < Z^*_{minZ}\right)$ or $\left(Z = Z^*_{minZ}\right.$ and $\left.K < K^*_{minZ}\right)$ then // new least cost incumbent //
        $Z^*_{minZ} = Z$;
        $K^*_{minZ} = K$;
        $\boldsymbol{T}^*_{minZ} = \boldsymbol{T}$;
    if $\left(K < K^*_{minK}\right)$ or $\left(K = K^*_{minK}\right.$ and $\left.Z < Z^*_{minK}\right)$ then // new least number of vehicles
     incumbent //
        $Z^*_{minK} = Z$;
        $K^*_{minK} = K$;
        $\boldsymbol{T}^*_{minK} = \boldsymbol{T}$
    if $\left(Z < Z^*_{K_{\text{UB}}}\right)$ and $\left(K \leq K_{\text{UB}}\right)$ then // new least cost incumbent meeting fleet size limit, if
     applicable //
        $Z^*_{K_{\text{UB}}} = Z$;
        $\boldsymbol{T}^*_{K_{\text{UB}}} = \boldsymbol{T}$;
    $t = t + 1$;
return $(\boldsymbol{T}^*_{minZ}, \boldsymbol{T}^*_{minK}, \boldsymbol{T}^*_{K_{\text{UB}}})$
```

4.A.2 Path-Scanning algorithm

Algorithm 4.2: *Path-Scanning*

```
Output: MCARPTIF solution $\boldsymbol{T}$
$\boldsymbol{R}_s = \boldsymbol{R}$ // all required arcs are initially unserviced //;
$\boldsymbol{T} = \Big[[[\sigma]]\Big]$ // solution initially consists of a single route with single subtrip //;
$u = \sigma, i = 1, j = 1$;
while $\boldsymbol{R}_s \neq \varnothing$ do // while there remains unserviced required arcs //
    $\boldsymbol{P}^{(i)} = \{x \in \boldsymbol{R}_s : Z(\boldsymbol{T}_i) + D(u,x) + w(x) + \mu^*(x,\sigma) \leq L\}$ // feasible arc additions are first isolated //;
    $\boldsymbol{P}^{(i,j)} = \{x \in \boldsymbol{P}^{(i)} : load(\boldsymbol{T}_{i,j}) + q(x) \leq Q\}$;
    $d^* = \begin{cases} \min\{D(u,x) : x \in \boldsymbol{P}^{(i,j)}\} \text{ if } \boldsymbol{P}^{(i,j)} \neq \varnothing, \\ \infty \quad \text{otherwise} \end{cases}$;
    $\boldsymbol{P}^{(i)}_{\text{IF}} = \{x \in \boldsymbol{R}_s : Z(\boldsymbol{T}_i) + \mu^*(u,x) + w(x) + \mu^*(x,\sigma) + \lambda \leq L\}$;
    $d^*_{\text{IF}} = \begin{cases} \min\{\mu^*(u,x) : x \in \boldsymbol{P}^{(i)}_{\text{IF}}\} \text{ if } \boldsymbol{P}^{(i)}_{\text{IF}} \neq \varnothing, \\ \infty \quad \text{otherwise} \end{cases}$;
    if $\boldsymbol{P}^{(i,j)} \neq \varnothing$ and $d^* \leq d^*_{\text{IF}}$ then // nearest arc is added to current subtrip //
        $\boldsymbol{S} = \{x \in \boldsymbol{P}^{(i,j)} : D(u,x) = d^*\}$;
        Select $v$ randomly from $\boldsymbol{S}$ // can be replaced with other tie-break rules //;
        $\boldsymbol{T}_{i,j} = \boldsymbol{T}_{i,j} \cup [v]$ // arc added to end of subtrip //;
        $\boldsymbol{R}_s = \boldsymbol{R}_s \setminus \{v, inv(v)\}$ // arc is removed from set of arcs requiring service //;
        $u = v$
    else if $\boldsymbol{P}^{(i)}_{\text{IF}} \neq \varnothing$ then// new subtrip is added with nearest arc after IF visit //
        $\boldsymbol{S}_{\text{IF}} = \{x \in \boldsymbol{P}^{(i)}_{\text{IF}} : \mu^*(u,x) = d^*_{\text{IF}}\}$;
        Select $v$ randomly from $\boldsymbol{S}_{\text{IF}}$ // can be replaced with other tie-break rules //;
        $\boldsymbol{T}_{i,j} = \boldsymbol{T}_{i,j} \cup [\Phi^*_{u,v}]$ // IF visit added to end of subtrip //;
        $\boldsymbol{T}_i = \boldsymbol{T}_i \cup \big[[\Phi^*_{u,v}, v]\big]$ // subtrip added to end of route //;
        $\boldsymbol{R}_s = \boldsymbol{R}_s \setminus \{v, inv(v)\}$ // arc is removed from set of arcs requiring service //;
        $j = j + 1, u = v$;
    else // vehicle returns to the depot and a new route is opened //
        $\boldsymbol{T}_{i,j} = \boldsymbol{T}_{i,j} \cup [\Phi^*_{u,\sigma}, \sigma]$ // IF and depot added to end of subtrip //;
        $\boldsymbol{T} = \boldsymbol{T} \cup \Big[[[\sigma]]\Big]$ // new route with one subtrip added to solution //;
        $i = i + 1, j = 1, u = \sigma$;
    if $\boldsymbol{R}_s = \varnothing$ then // all arcs are serviced //
        $\boldsymbol{T}_{i,j} = \boldsymbol{T}_{i,j} \cup [\Phi^*_{u,\sigma}, \sigma]$ // IF and depot added to end of subtrip //;
return $(\boldsymbol{T})$
```

4.A.3 Randomised-Merge algorithm

Algorithm 4.3: *Initialise-Merge*

```
Output: Initial partial solution $\boldsymbol{T}^{\mathrm{P}}$ (without dummy-arcs), arc to solution mapping $T^{-1}$, ordered
        merge savings list $\boldsymbol{M}$, and merge saving to arc pair mappings, $\boldsymbol{M}_{\rightarrow}$ and $\boldsymbol{M}_{\Rightarrow}$.
$i = 0$;
for $u \in \boldsymbol{R}$ do
    $i = i + 1$;
    $\boldsymbol{T}^{\mathrm{P}}_i = [[u]]$ // route is created for each required arc //;
    $T^{-1}(u) = i$ // pointer function is updated that maps an arc to its route's index //;
$\boldsymbol{M}_{\rightarrow} = \varnothing, \boldsymbol{M}_{\Rightarrow} = \varnothing, \boldsymbol{M}^{-1}_{\rightarrow} = \varnothing, \boldsymbol{M}^{-1}_{\Rightarrow} = \varnothing$;
for $u \in \boldsymbol{R}$ do
    for $v \in \boldsymbol{R} \setminus \{u, inv(u)\}$ do
        $m' = \mu^*(u,\sigma) + D(\sigma,v) + \lambda - D(u,v) - p(u,v)$ // direct merge savings, including
          orientation penalty //;
        if $m' \notin \boldsymbol{M}_{\rightarrow}$ then
            $\boldsymbol{M}_{\rightarrow} = \boldsymbol{M}_{\rightarrow} \cup \{m'\}$ // saving added to direct savings list //;
            $\boldsymbol{M}^{-1}_{\rightarrow}(m') = \{(u,v)\}$ // pointer function is updated that maps savings to arcs to be
              directly merged //;
        else // cost saving is equal to that of other mergers //
            $\boldsymbol{M}^{-1}_{\rightarrow}(m_{\rightarrow}) = \boldsymbol{M}^{-1}_{\rightarrow}(m_{\rightarrow}) \cup \{(u,v)\}$;
        $m'' = \mu^*(u,\sigma) + D(\sigma,v) - \mu^*(u,v) - p(u,v)$ // IF merge savings, including orientation
          penalty //;
        if $m'' \notin \boldsymbol{M}_{\Rightarrow}$ then
            $\boldsymbol{M}_{\Rightarrow} = \boldsymbol{M}_{\Rightarrow} \cup \{m''\}$ // saving added to indirect savings list //;
            $\boldsymbol{M}^{-1}_{\Rightarrow}(m'') = \{(u,v)\}$ // pointer function is updated that maps savings to arcs to be
              merged through IFs //;
        else // cost saving is equal to that of other mergers //
            $\boldsymbol{M}^{-1}_{\Rightarrow}(m'') = \boldsymbol{M}^{-1}_{\Rightarrow}(m'') \cup \{(u,v)\}$;
$\boldsymbol{M} = \boldsymbol{M}_{\rightarrow} \cup \boldsymbol{M}_{\Rightarrow}$;
Sort $\boldsymbol{M}$ in nonincreasing order $\boldsymbol{M} = [M_1, \ldots, M_{|\boldsymbol{M}|}]$, such that $M_i > M_{i+1} \, \forall \, i \in \{1, \ldots, |\boldsymbol{M}|\}$;
return $(\boldsymbol{T}^{\mathrm{P}}, T^{-1}, \boldsymbol{M}, \boldsymbol{M}^{-1}_{\rightarrow}, \boldsymbol{M}^{-1}_{\Rightarrow})$
```

Algorithm 4.4: *Execute-Merge* (without symmetrical route inversion)

Input : Initial partial solution $\boldsymbol{T}^{\mathrm{P}}$ (without dummy-arcs), arc to solution mapping T^{-1}, ordered merge savings list $\boldsymbol{M}$, and merge saving to arc pair mappings, $\boldsymbol{M}_{\to}$ and $\boldsymbol{M}_{\Rightarrow}$.
Output: Partial solution $\boldsymbol{T}^{\mathrm{P}}$ (without dummy-arcs).

```
for $m \in M$ do // merge starts with the greatest merge saving //
    while $\boldsymbol{M}_{\to}^{-1}(m) \cup \boldsymbol{M}_{\Rightarrow}^{-1}(m) \neq \varnothing$ do // there are potentially feasible mergers to implement with
      savings $m$ //
        $\boldsymbol{M}'_{\to} = \boldsymbol{M}_{\to}^{-1}(m), \boldsymbol{M}'_{\Rightarrow} = \boldsymbol{M}_{\Rightarrow}^{-1}(m)$ // sets of all potential direct and IF mergers with equal
          savings $m$ //;
        for $(u,v) \in \boldsymbol{M}_{\to}^{-1}(m) \cup \boldsymbol{M}_{\Rightarrow}^{-1}(m)$ do // for each potential merger, eliminate infeasible ones
          //
            $i = T^{-1}(u), j = T^{-1}(v)$ // find route indices of arc routes //;
            if $i = 0$ or $j = 0$ then
                $merge = False$ // arc is no longer part of solutions, as its inverse is included //;
            else if $u \neq \boldsymbol{T}^{\mathrm{P}}_{i,|\boldsymbol{T}_i|}$ or $v \neq \boldsymbol{T}^{\mathrm{P}}_{j,1}$ then
                $merge = False$ // arc u and v are not the last and first arcs in a route //;
            else if $i = j$ then
                $merge = False$ // arcs belong to the same route //
            else if $z(u) + z(v) - m - p(u,v) > L$ then
                $merge = False$ // cost of merged route exceeds route cost limit //;
            else if $(u,v) \in \boldsymbol{M}_{\Rightarrow}^{-1}(m)$ and $load\big(\boldsymbol{T}^{\mathrm{P}}_{i,|\boldsymbol{T}^{\mathrm{P}}_i|}\big) + load\big(\boldsymbol{T}_{j,1}\big) > Q$ then
                $merge = False$ // demand of merged subtrip exceeds vehicle capacity //;
            else
                $merge = True$ // merge is feasible //;
            if $merge = False$ and $(u,v) \in \boldsymbol{M}_{\to}^{-1}(m)$ then
                $\boldsymbol{M}_{\to}^{-1}(m) = \boldsymbol{M}_{\to}^{-1}(m) \setminus \{(u,v)\}$ // remove merge from set of potential direct
                  mergers //;
            else if $merge = False$ and $(u,v) \in \boldsymbol{M}_{\Rightarrow}^{-1}(m)$ then
                $\boldsymbol{M}_{\Rightarrow}^{-1}(m) = \boldsymbol{M}_{\Rightarrow}^{-1}(m) \setminus \{(u,v)\}$ // remove merge from set of potential IF mergers
                  //;
        if $\boldsymbol{M}_{\to}^{-1}(m) \cup \boldsymbol{M}_{\Rightarrow}^{-1}(m) \neq \varnothing$ then // there are mergers to implement //
            Randomly choose $(u,v)$ from $\boldsymbol{M}'_{\to}(m) \cup \boldsymbol{M}'(m)_{\Rightarrow}$ // can be replaced with other
              tie-break rules //;
            $i = T^{-1}(u), j = T^{-1}(v)$ // find route indices of arc routes //;
            if $(u,v) \in \boldsymbol{M}_{\to}^{-1}(m)$ then // merge belongs to direct merge set //
                $\boldsymbol{T}^{\mathrm{P}}_{i,|\boldsymbol{T}_i|} = \boldsymbol{T}^{\mathrm{P}}_{i,|\boldsymbol{T}_i|} \cup \boldsymbol{T}^{\mathrm{P}}_{j,1}$ // combine last and first subtrips //;
                $\boldsymbol{T}^{\mathrm{P}}_i = \boldsymbol{T}^{\mathrm{P}}_i \cup \boldsymbol{T}^{\mathrm{P}}_j \setminus \boldsymbol{T}^{\mathrm{P}}_{j,1}$ // combine rest of routes //;
            else if $(u,v) \in \boldsymbol{M}_{\Rightarrow}^{-1}(m)$ then // merge belongs to IF merge set //
                $\boldsymbol{T}^{\mathrm{P}}_i = \boldsymbol{T}^{\mathrm{P}}_i \cup \boldsymbol{T}^{\mathrm{P}}_j$ // combine two routes //;
            $\boldsymbol{T}^{\mathrm{P}}_j = \varnothing$ // route $j$ is removed from the solution since all its arcs are in route $i$ //;
            $T^{-1}(T^{\mathrm{P}}_{i,|\boldsymbol{T}^{\mathrm{P}}_i|,|\boldsymbol{T}^{\mathrm{P}}_{i,|\boldsymbol{T}^{\mathrm{P}}_i|}|}) = i$ // route pointer function is updated so that the last arc in
              the new route also points to $i$ //;
            if $inv(u) \neq 0$ and $T^{-1}(inv(u)) \neq 0$ then
                $i' = T^{-1}(inv(u))$;
                $\boldsymbol{T}^{\mathrm{P}}_{i'} = \varnothing$ // apposing single-arc route is removed from solution //;
                $T^{-1}(inv(u)) = 0$ // apposing arc cannot be merged into other routes //;
            if $inv(v) \neq 0$ and $T^{-1}(inv(v)) \neq 0$ then
                $j' = T^{-1}(inv(v))$;
                $\boldsymbol{T}^{\mathrm{P}}_{j'} = \varnothing$ // apposing single-arc route is removed from solution //;
                $T^{-1}(inv(v)) = 0$ // apposing arc cannot be merged into other routes //;
return $(\boldsymbol{T}^{\mathrm{P}})$
```

4.A.4 Reduce-Vehicles algorithm

Algorithm 4.5: *Insert-Arc*

```
Input   : Incomplete solution $\boldsymbol{T}$, and arc to insert $u$.
Output: Solution $\boldsymbol{T}$ with $u$ possibly inserted.
$i^* = j^* = n^* = 0$;
$m^* = \infty$;
for $i = 1$ to $|\boldsymbol{T}|$ do
    for $j = 1$ to $|\boldsymbol{T}_i|$ do
        if $j = |\boldsymbol{T}_i|$ then
            $J = |\boldsymbol{T}_{i,j}| - 1$ // IF and depot visits are fixed at the end of routes //;
        else
            $J = |\boldsymbol{T}_{i,j}|$ // IF visit is fixed at end of route //;
        for $k = 2$ to $J$ do
            $m = D(T_{i,j,n}, u) + w(u) + D(u, T_{i,j,n+1}) - D(T_{i,j,n}, T_{i,j,n+1})$ // cost of arc insert //;
            if $Z(\boldsymbol{T}_{i,j}) + q(u) \leq Q$ and $\boldsymbol{Z}(T_i) + m \leq L$ then // check that arc insert is feasible //
                if $m \leq m^*$ then // new best insert position found //
                    $m^* = m$;
                    $i^* = i$;
                    $j^* = j$;
                    $n^* = n$;
if $i^* \neq 0$ then // insert arc into best feasible position //
    $\boldsymbol{T}_{i^*,j^*} = [T_{i^*,j^*,1}, \dots, T_{i^*,j^*,n^*}] \cup [u] \cup [T_{i^*,j^*,n^*+1}, \dots, T_{i^*,j^*,|\boldsymbol{T}_{i^*,j^*}|}]$;
else // no feasible insert position could be found //
    $\boldsymbol{T} = \varnothing$;
return ($\boldsymbol{T}$)
```

Algorithm 4.6: *Reduce-Vehicles*

```
Input  : Solution $\boldsymbol{T}$ with $K$ routes
Output: Solution $\boldsymbol{T}^*$ with $K$ possibly reduced
Let $\boldsymbol{T}^* = \boldsymbol{T}$, and sort $\boldsymbol{T}^*$ such that $Z(\boldsymbol{T}^*_i) \leq Z(\boldsymbol{T}^*_{i+1}) \,\forall\, i \in \{1, \ldots, K\}$;
$K = |\boldsymbol{T}^*|$;
$i = 1$ // start with least cost route. //;
while $i \leq K$ do
    $\boldsymbol{S} = \{u \in \boldsymbol{T}^*_{i,1} \cup \boldsymbol{T}^*_{i,2}, \ldots, \boldsymbol{T}^*_{i,|\boldsymbol{T}^*_i|}\}$;
    $\boldsymbol{T} = \boldsymbol{T}^* \setminus \boldsymbol{T}^*_i$ // remove route from solution //;
    Sort $\boldsymbol{S}$ in decreasing order according the tasks demands;
    $n = 1$ // start with highest demand arc. //;
    while $n \leq |\boldsymbol{S}|$ do
        $u = \boldsymbol{S}_n$;
        $\boldsymbol{T}$ =Insert-Arc$(\boldsymbol{T}, u)$ // insert arc in the best position using Algorithm 4.5 //;
        if $\boldsymbol{T} = \varnothing$ then
            $n = |\boldsymbol{S}| + 1$ // no feasible insertion could be found //;
        else
            $n = n + 1$ // arc was successfully inserted and next lowest demand arc will be
              evaluated //;
    if $\boldsymbol{T} \neq \varnothing$ then // number of vehicles was successfully reduced //
        $\boldsymbol{T}^* = \boldsymbol{T}$;
        $K = |\boldsymbol{T}^*|$;
        $i = 1$ // process is repeated //;
    else
        $i = i + 1$ // try next least cost route //;
return $(\boldsymbol{T}^*)$
```

Chapter 5

Basic local search heuristics

Constructive heuristics are capable of quickly generating feasible solutions for $\mathscr{NP}$-hard combinatorial optimisation problems. They are well suited when near real-time decision support is required, but their solutions leave room for improvement. When more execution time is available, the constructive heuristic solutions can be further improved using Local Search (LS) strategies. In this chapter we develop five LS move operators for the Mixed Capacitated Arc Routing Problem under Time Restrictions with Intermediate Facilities (MCARPTIF) and test four basic LS implementations on the problem. Two widely used methods were applied to to speed-up LS, but despite their application LS proved too slow for medium-term planning on large waste collection instances. This necessitated us to develop more advanced LS acceleration mechanisms, which we present in the next chapter.

5.1 Introduction

LS, introduced in the 1960s, is a widely applied improvement method that starts with an initial solution and progressively moves to an improving neighbour solution until a local optimum is reached [92]. It is also used as the basic optimisation component of more intelligent metaheuristic strategies, such as Guided Local Search, Variable Neighbourhood Search, and Memetic Algorithms [80]. Metaheuristics for the Capacitated Arc Routing Problem (CARP), Mixed Capacitated Arc Routing Problem (MCARP) and the Capacitated Arc Routing Problem under Time Restrictions with Intermediate Facilities (CARPTIF) also rely heavily on LS. In fact, all recent metaheuristics for the three problems, reviewed by Prins [69] and Muyldermans and Pang [64], rely on some form of LS, making it an important research area for waste collection applications.

The main components of LS are its move operators that allow it to move to improving solutions. In the next section we review move operators that have been developed for the MCARP, as well as some of the most widely used, yet basic LS acceleration mechanisms. In Section 5.3 we present the move operators and basic LS components that we developed for the MCARPTIF. Computational results on waste collection instances for four LS implementations are presented in Section 5.4. The aim of the tests was to evaluate the general improvement capabilities of LS, and to determine if the basic implementations could be used as-is for either short or medium-term planning. The chapter is then concluded in Section 5.5.

5.2 Local search for the CARP and MCARP

The basic LS template, adapted from Irnich et al. [48], is shown in Algorithm 5.1. 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, $\mathscr{N}$, is a mapping $\mathscr{N} : \boldsymbol{X} \to 2^{\boldsymbol{X}}$, and each element $\boldsymbol{T}' \in \mathscr{N}(\boldsymbol{T})$ 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, LS replaces the current solution $\boldsymbol{T}^{(t)}$ by an improving neighbour $\boldsymbol{T}^{(t+1)} \in \mathscr{N}(\boldsymbol{T}^{(t)})$. The search terminates when a local optimum is reached, meaning there are no improving neighbours in $\mathscr{N}(\boldsymbol{T}^{(t)})$.

Algorithm 5.1: *Generic-Local-Search*

```
Input  : An initial solution $\boldsymbol{T}^{(0)} \in \boldsymbol{X}$.
Output: Local optimum $\boldsymbol{T}^{(t)}$.
$t = 0$;
repeat
    Generate $\mathscr{N}(\boldsymbol{T}^{(t)})$;
    Search for an improving neighbour $\boldsymbol{T}' \in \mathscr{N}(\boldsymbol{T}^{(t)})$;
    if an improving neighbour was found then
        $\boldsymbol{T}^{(t+1)} = \boldsymbol{T}'$;
        $t = t + 1$;
until an improving neighbour could not be found;
return $(\boldsymbol{T}^{(t)})$
```

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 the CARP and MCARP, moves typically change the service position of required arcs or edges between and within routes. Belenguer et al. [6] and Lacomme et al. [51] use seven move operators for their MCARP LS implementations, embedded within Memetic Algorithms. The seven operators are discussed next, followed by a review of a few basic LS acceleration mechanisms.

5.2.1 MCARP move operators

Consistent with the notation from previous chapters, an MCARP solution, $\boldsymbol{T}$, is a list, $[\boldsymbol{T}_1, \ldots, \boldsymbol{T}_{|\boldsymbol{T}|}]$, of $|\boldsymbol{T}|$ vehicle routes. Each route, $\boldsymbol{T}_i$, consists of a list of arc tasks $[T_{i,1}, \ldots, T_{i,|\boldsymbol{T}_i|}]$. A pointer function, $inv(u) = v$, is used to return the opposing arc of u where u and v represent the two directions of an edge in the original graph, $\boldsymbol{G}$. If u represents an arc in $\boldsymbol{G}$ the pointer returns $inv(u) = 0$. Let $\boldsymbol{R}$ be the set of all required arc tasks and let $\boldsymbol{R}_T \subseteq \boldsymbol{R}$ be the set of arcs that is currently in solution $\boldsymbol{T}$, excluding dummy arcs. It thus consists of all the arc tasks with $inv(u) = 0$, and one of either u or v where $inv(u) = v$ and $inv(u) \neq 0$. Lastly, let $T_{i,j} = u$ and $T_{l,m} = v$ be two different tasks, which can be in the same or different routes.

Five of the seven MCARP move operators of Lacomme et al. [51] and Belenguer et al. [6] are shown in Figure 5.1. 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 the figure, with adjacent arcs, u and $T_{i,j+1}$, moved together to a new position. It is thus not valid when u is the last task in a route. The *exchange*

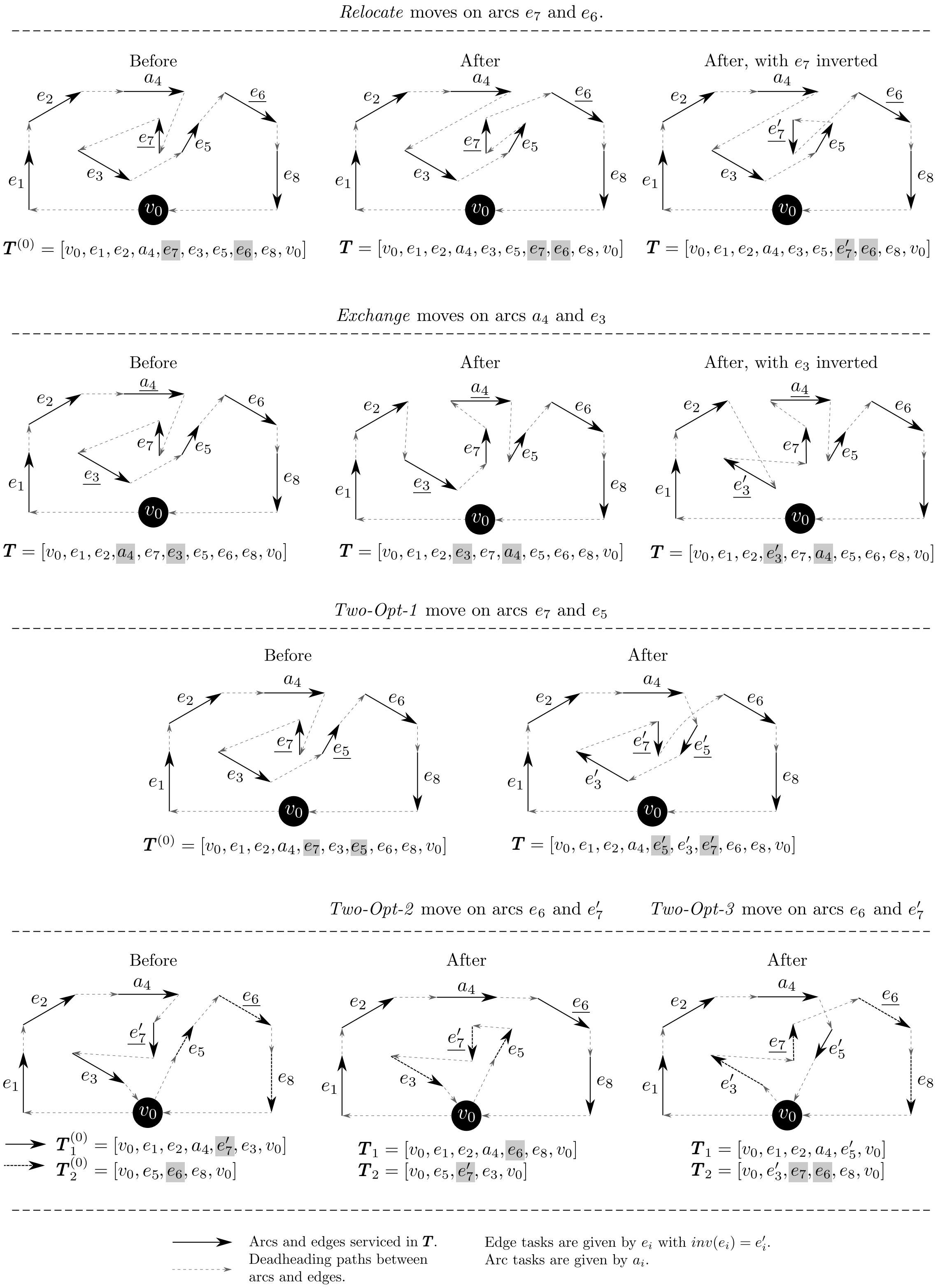

Relocate moves on arcs e_7 and e_6.
Before
After
After, with e_7 inverted
a_4
e_1
e_2
e_3
e_5
e_6
e_7
e_8
v_0
$\boldsymbol{T}^{(0)} = [v_0, e_1, e_2, a_4, e_7, e_3, e_5, e_6, e_8, v_0]$
$\boldsymbol{T} = [v_0, e_1, e_2, a_4, e_3, e_5, e_7, e_6, e_8, v_0]$
$\boldsymbol{T} = [v_0, e_1, e_2, a_4, e_3, e_5, e'_7, e_6, e_8, v_0]$
Exchange moves on arcs a_4 and e_3
Before
After
After, with e_3 inverted
$\boldsymbol{T} = [v_0, e_1, e_2, a_4, e_7, e_3, e_5, e_6, e_8, v_0]$
$\boldsymbol{T} = [v_0, e_1, e_2, e_3, e_7, a_4, e_5, e_6, e_8, v_0]$
$\boldsymbol{T} = [v_0, e_1, e_2, e'_3, e_7, a_4, e_5, e_6, e_8, v_0]$
Two-Opt-1 move on arcs e_7 and e_5
Before
After
$\boldsymbol{T}^{(0)} = [v_0, e_1, e_2, a_4, e_7, e_3, e_5, e_6, e_8, v_0]$
$\boldsymbol{T} = [v_0, e_1, e_2, a_4, e'_5, e'_3, e'_7, e_6, e_8, v_0]$
Two-Opt-2 move on arcs e_6 and e'_7
Two-Opt-3 move on arcs e_6 and e'_7
Before
After
After
$\boldsymbol{T}_1^{(0)} = [v_0, e_1, e_2, a_4, e'_7, e_3, v_0]$
$\boldsymbol{T}_2^{(0)} = [v_0, e_5, e_6, e_8, v_0]$
$\boldsymbol{T}_1 = [v_0, e_1, e_2, a_4, e_6, e_8, v_0]$
$\boldsymbol{T}_2 = [v_0, e_5, e'_7, e_3, v_0]$
$\boldsymbol{T}_1 = [v_0, e_1, e_2, a_4, e'_5, v_0]$
$\boldsymbol{T}_2 = [v_0, e'_3, e_7, e_6, e_8, v_0]$
Arcs and edges serviced in $\boldsymbol{T}$.
Deadheading paths between arcs and edges.
Edge tasks are given by e_i with $inv(e_i) = e'_i$.
Arc tasks are given by a_i.

Figure 5.1: Examples of *relocate*, *exchange*, *two-opt* and *two-opt-1* move operators for the MCARP.

operator, shown in the figure, exchanges the positions of two arcs, u and v, so that $T_{i,j} = v$ and $T_{l,m} = u$. The third operator, *flip*, inverts u so that $T_{i,j} = inv(u)$ if $inv(u) \neq 0$. A more advanced version of the operator is used by Beullens et al. [10] in which the optimal orientation of all arcs in a route is efficiently determined.

The rest of the MCARP operators employ *two-opt* moves that first delete links between tasks $T_{i,j-1}$ and u, and $T_{l,m+1}$ and v. Next, the resulting route segments are relinked 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 5.1. *Two-opt-2* is more intuitively referred to as *cross* by Beullens et al. [10] 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. *Two-opt-1* and *two-opt-3* involve the reversal of certain route segments to be relinked. For the symmetrical CARP this can be automatically done without any additional calculations. The same does not always hold for the asymmetrical MCARP. For an MCARP route segment to be symmetrical, all its arc tasks must have inverse tasks, and the deadheading time between two tasks must be the same 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. [51] and Belenguer et al. [6] discard *two-opt* moves if the segments contain any arc-task u with $inv(u) = 0$. The example *two-opt-1* move in Figure 5.1 will thus be allowed, but the *two-opt-3* move will be discarded.

Except for *flip*, move operators are applied to all combinations of $\{u, v\} \in \boldsymbol{R}_T : u \neq v$. For *exchange* and *two-opt* moves the condition $u < v$ may be enforced, since the move between u and v is to the same as the one between v and u. 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. [51] and Belenguer et al. [6] use a *first-move* strategy.

Belenguer et al. [6] attempt to improve the efficiency of their LS implementation by forcing *two-opt* to discard moves involving asymmetrical route segments, and by using the more efficient *first-move* strategy. Despite these initiatives they found that the execution time of their Memetic Algorithm is excessive on the biggest of their test instances, instances that are consistent in size with real-world applications. For future work they recommend using advanced acceleration mechanisms to speed-up the LS component of their Memetic Algorithm. In this chapter we only focus on and adapt a few basic accelerate mechanisms, with the more advanced mechanisms being the subject of the next chapter.

5.2.2 Basic acceleration strategies

Most of the computational time spent within LS is on scanning the neighbourhood for improving moves in each iteration. For the CARP and MCARP the most used acceleration method is *first-move* instead of *best-move*. The main differences between the strategies are illustrated in Section 5.A, at the end of the chapter. 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. In this case the complete neighbourhood will be scanned in a futile effort to find an improving move. *First-move* is used in the majority of LS implementations including those in [6, 51] for the MCARP, and in [10, 12] for the CARP.

The solution quality and efficiency trade-off between *best-move* and *first-move* is investigated by Santos et al. [78] for the CARP. During preliminary testing on *bccm*, *val*, *eglese* and *bmcv* instances[1], they found that *best-move* employing twelve move operators is much slower than *first-move*, but that it terminates at better (lower cost) local optima. The authors further test a Variable Neighbourhood Search (VNS) implementation. For VNS, distinct neighbourhoods are defined, each resulting from a different move operator. The neighbourhoods are scanned in a predefined order in each iteration. Starting from the first neighbourhood, the best improving move is found and made. If none is found the next neighbourhood is scanned. Once the best move from a neighbourhood is made, the process repeats starting from the first neighbourhood. Neighbourhoods are scanned in order of their move operator's performance ranking, from best to worst, which the authors establish from computational tests with the *best-move* strategy. Santos et al. [78] state that the VNS version does not match *best-move* on solution quality but it is much faster, with execution times close to that of *first-move.* They further found that VNS terminates at better local optima than *first-move.*

Due to a lack of formal testing between *best-move*, *first-move* and VNS on realistically sized instances it is difficult to predict which strategy will perform the best on the MCARPTIF. To limit the scope of this thesis, we implemented and tested *best-move* and *first-move* strategies for the MCARPTIF, and leave their evaluation against VNS for future work.

In most LS implementations, including those of Belenguer et al. [6] and Lacomme et al. [51], a combination of five main move operators are used. Using more move operators, such as the twelve of Santos et al. [78], increases the size of the LS neighbourhood. As a result, LS takes more time to partially or fully scan the neighbourhood. The efficiency of LS could be improved by identifying and eliminating operators that have little impact on solution quality. Such an analysis has yet to be completed for the MCARP. Santos et al. [78] did informal analysis on the CARP by ranking their twelve move operators and testing two versions of their metaheuristic: the first using LS with only the top six ranked moves, and the second with all twelve moves. As expected, the twelve move operator version was slower but performed better. In this thesis, to identify non-critical move operators for elimination, we performed a similar test by critically evaluating the contributions of move operators within the *best-move* framework. The analysis was then used to speed-up our LS implementations by eliminating non-essential operators.

5.3 Basic local search for the MCARPTIF

The following MCARPTIF terminology is used throughout this chapter. 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 list of arc tasks $[T_{i,j,1}, \ldots, T_{i,j,|\boldsymbol{T}_{i,j}|}]$. The service time and demand of arc u are given by $w(u)$ and $q(u)$, respectively. The pointer function, $inv(u) = v$, is used to return the opposing arc of u. The travel time for the shortest path from arc u to arc v is given by $D(u, v)$ and it is assumed that the shortest path is always followed between consecutive tasks. The best Intermediate Facility (IF) to

[1] Santos et al. [78] do not specify which instances were used for their preliminary tests. Detailed results are presented on the specified CARP instances, so we assumed they were used for the preliminary tests as well.

visit after servicing arc u and before servicing arc v is pre-calculated and given by $\Phi^*(u,v)$. For our LS implementation, the duration of the visit, *including* offloading time, λ, is given by $\mu^*(u,v)$. It is assumed that the best IF is always visited between two arcs and between an arc and the depot, given as σ. The load of a subtrip is given by $load(\boldsymbol{T}_{i,j})$ and it may not exceed the vehicle capacity limit, Q. The cost of route $\boldsymbol{T}_i$ is given as $Z(\boldsymbol{T}_i)$ and the cost of the solution is given as $Z(\boldsymbol{T})$. Again, let $\boldsymbol{R}$ be the set of all required arc tasks and let $\boldsymbol{R}_T \subseteq \boldsymbol{R}$ be the set of arcs that is currently in solution $\boldsymbol{T}$, excluding dummy arcs. Lastly, let $T_{i,j,k} = u$ and $T_{l,m,n} = v$ be two different tasks, which can be in the same or different routes or subtrips.

An important modification that we made to the MCARPTIF solution representation from the previous chapters is that the last IF and depot visit tasks are included in a separate subtrip at the end of the route, such that the $\boldsymbol{T}_i$ ends with subtrip $\boldsymbol{T}_{i,j} = [\Phi^*(T_{i,j-1,|\boldsymbol{T}_{i,j-1}|-1},\sigma),\sigma]$, where $j = |\boldsymbol{T}_i|$. This simplifies the LS implementations by allowing the second last subtrip,

$$\boldsymbol{T}_{i,j-1} = [\ldots, T_{i,j-1,|\boldsymbol{T}_{i,j-1}|-2}, T_{i,j-1,|\boldsymbol{T}_{i,j-1}|-1}, \Phi^*(T_{i,j-1,|\boldsymbol{T}_{i,j-1}|-1},\sigma)],$$

now without the depot visit, to be treated the same as its preceding subtrips. If this modification is not applied, special checks and procedures would have to be developed for moves involving the last subtrips in routes. Lastly, to make our illustrations more concise we use Φ' to denote the best IF visit between the last and first arc 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

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

The main framework of the basic LS heuristics that we developed for the MCARPTIF is shown in Algorithm 5.2. At the start of each LS iteration *Reduce-Vehicles*, developed in the previous chapter, was applied in an attempt to reduce the vehicle fleet size. It may be

Algorithm 5.2: *Basic-Local-Search*

```
Input  : An initial solution $\boldsymbol{T}^{(0)} \in \boldsymbol{X}$.
Output: Local optimum $\boldsymbol{T}^{(t)}$
1  $t = 0$;
2  repeat
3      Use Reduce-Vehicles on $\boldsymbol{T}^{(t)}$ to reduce the fleet, and let $\boldsymbol{T}'$ be the result;
4      if $|\boldsymbol{T}'| < |\boldsymbol{T}^{(t)}|$ then // the fleet size has been reduced //
5          Set $\boldsymbol{T}^{(t)} = \boldsymbol{T}'$;
6          // if min $Z$ is the primary objective, the condition in line 5 can be replaced by
             $Z(\boldsymbol{T}') < Z(\boldsymbol{T}^{(t)})$ //;
7      Use Efficient-IF-Split (Algorithm 3.1 in Chapter 3) on $\boldsymbol{T}^{(t)}$ to improve the IF visit placements
         of the routes, and let $\boldsymbol{T}'$ be the result;
8      On $\boldsymbol{T}'$, use the flip, relocate, exhange, cross and two-opt-1 move operators (Algorithms 5.3
         to 5.11) to find and return the overall best feasible improving move;
9      // for first-move, the move operators are used to return the first improving move found //;
10     if a feasible improving move was found then
11         Implement the move on $\boldsymbol{T}'$ and let $\boldsymbol{T}''$ be the result;
12         Set $\boldsymbol{T}^{(t+1)} = \boldsymbol{T}''$;
13         $t = t + 1$;
14 until a feasible improving move could not be found;
15 return ($\boldsymbol{T}^{(t)}$)
```

applied even when minimising cost is the primary objective, in which case reduced fleet solutions should only be accepted if they have a lower total cost. Next the optimal IF splitting procedure for the Mixed Capacitated Arc Routing Problem with Intermediate Facilities (MCARPIF), *Efficient-IF-Split* developed in Chapter 3, was applied to improve the placement of IF visits within each route. Only thereafter did LS search for an improving move.

The heuristic relies on five move operators, referred to in line 8, namely *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. Implementation details for all the move operators are presented in the rest of this section.

5.3.1 Flip

The first move operator that we modify for the MCARPTIF is *flip*. As shown in Figure 5.2, *flip* changes the service orientation of an arc task u, should it have an opposing arc, i.e. $inv(u) \neq 0$. After a move involving arcs next to IFs are implemented, the IF tasks are automatically updated to the best IF visit, as given by Φ'. The cost of the move depends on the position of the arc relative to IF visits, with all possible cost formulas and the conditions for their application shown in Figure 5.3. Where the position of an arc is changed relative to an IF, the duration of the best IF visit, $\mu^*(u, v)$, is used for the cost calculations. Where the position is changed relative to an arc or depot task, the shortest dead-heading path time, $D(u, v)$, is used.

Original route $\boldsymbol{T}_i$ with a planned move to *flip* arc $T_{i,1,3} = u_2$:

$$\boldsymbol{T}_i = [[\sigma, u_1, u_2, u_3, \Phi'], [\Phi', u_4, u_5, \Phi'], [\Phi', u_6, \Phi'], [\Phi', \sigma]]$$

Route $\boldsymbol{T}'_i$ after the *flip* move has been implemented:

$$\boldsymbol{T}'_i = [[\sigma, \underline{u_1}, \underline{inv(u_2)}, \underline{u_3}, \Phi'], [\Phi', u_4, u_5, \Phi'], [\Phi', u_6, \Phi'], [\Phi', \sigma]]$$

Arcs affected by move

Figure 5.2: Example of the *flip* move where an arc task in a route is inverted if it has an opposing arc.

The location and cost of the best improving *flip* move for any subtrip $\boldsymbol{T}_{i,j}$ is returned by Algorithm 5.3. *Flip* does not change the demand serviced on subtrips, and since it only considers improving moves, *flip* will never result in the route duration limit being exceeded. As such, no feasibility checks are necessary as improving moves will always be feasible.

The algorithm structure can be easily modified for a *first-move* strategy. As shown in the comments of Algorithm 5.3, the algorithm simply stops the moment that a feasible improving move is found and returns the move's parameters. This modification applies to all the algorithms of this section. Algorithm 5.4 is used to find the best *flip* move (or first move as shown in the comment on line 10) among all routes and subtrips. During an LS iteration, the arc in each route and subtrip is scanned to find the best *flip* move, giving it a computational complexity of $\mathcal{O}(|\boldsymbol{R}_T|)$ per iteration.

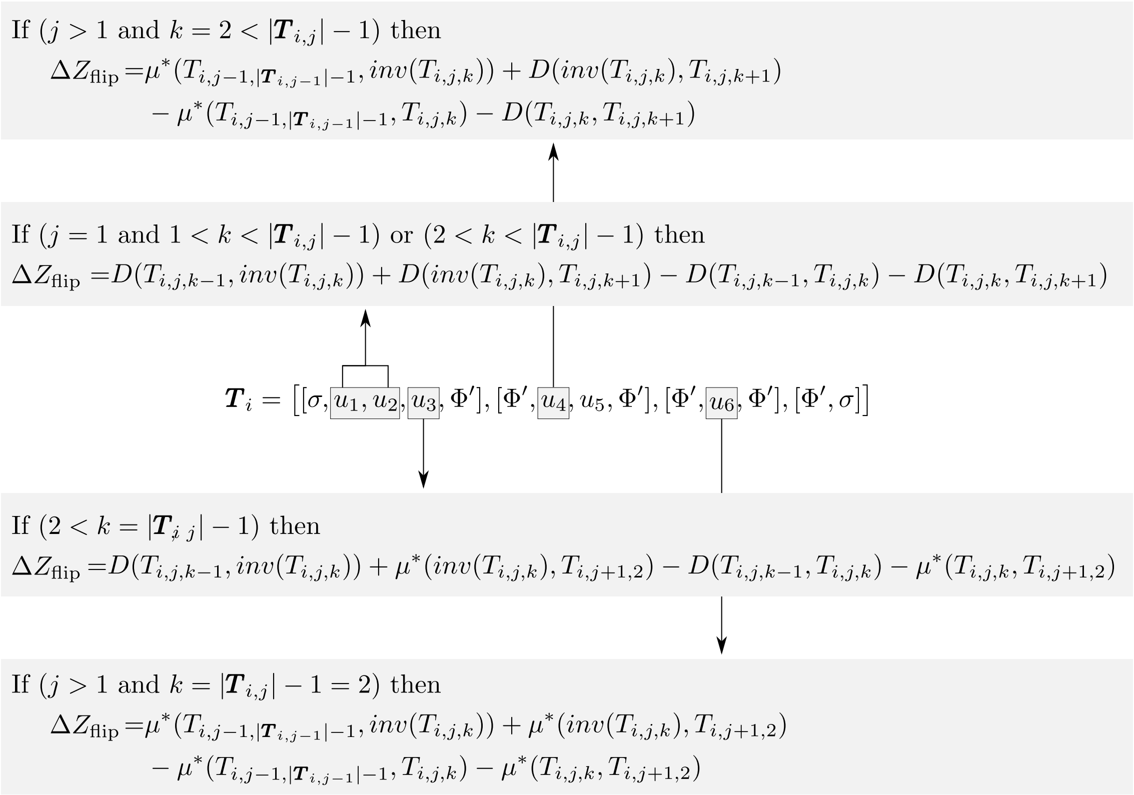

If $(j > 1$ and $k = 2 < |\boldsymbol{T}_{i,j}| - 1)$ then
$\Delta Z_{\text{flip}} = \mu^*(T_{i,j-1,|\boldsymbol{T}_{i,j-1}|-1}, inv(T_{i,j,k})) + D(inv(T_{i,j,k}), T_{i,j,k+1}) - \mu^*(T_{i,j-1,|\boldsymbol{T}_{i,j-1}|-1}, T_{i,j,k}) - D(T_{i,j,k}, T_{i,j,k+1})$
If $(j = 1$ and $1 < k < |\boldsymbol{T}_{i,j}| - 1)$ or $(2 < k < |\boldsymbol{T}_{i,j}| - 1)$ then
$\Delta Z_{\text{flip}} = D(T_{i,j,k-1}, inv(T_{i,j,k})) + D(inv(T_{i,j,k}), T_{i,j,k+1}) - D(T_{i,j,k-1}, T_{i,j,k}) - D(T_{i,j,k}, T_{i,j,k+1})$
$\boldsymbol{T}_i = [[\sigma, \boxed{u_1, u_2}, \boxed{u_3}, \Phi'], [\Phi', \boxed{u_4}, u_5, \Phi'], [\Phi', \boxed{u_6}, \Phi'], [\Phi', \sigma]]$
If $(2 < k = |\boldsymbol{T}_{i,j}| - 1)$ then
$\Delta Z_{\text{flip}} = D(T_{i,j,k-1}, inv(T_{i,j,k})) + \mu^*(inv(T_{i,j,k}), T_{i,j+1,2}) - D(T_{i,j,k-1}, T_{i,j,k}) - \mu^*(T_{i,j,k}, T_{i,j+1,2})$
If $(j > 1$ and $k = |\boldsymbol{T}_{i,j}| - 1 = 2)$ then
$\Delta Z_{\text{flip}} = \mu^*(T_{i,j-1,|\boldsymbol{T}_{i,j-1}|-1}, inv(T_{i,j,k})) + \mu^*(inv(T_{i,j,k}), T_{i,j+1,2}) - \mu^*(T_{i,j-1,|\boldsymbol{T}_{i,j-1}|-1}, T_{i,j,k}) - \mu^*(T_{i,j,k}, T_{i,j+1,2})$

Figure 5.3: Four different possible costs of *flipping* an arc in subtrip $\boldsymbol{T}_{i,j}$. The cost depends on the arc's position, $T_{i,j,k}$, relative to intermediate facility visits.

Algorithm 5.3: *Find-Flip-Subtrip-Move*

```
Input  : Subtrip $\boldsymbol{T}_{i,j}$ on which the move will be applied.
Output: Cost-saving, $\Delta Z^*$; and arc position, $k^*$, of the best subtrip move.
$\Delta Z^* = 0$;
$k^* = 0$;
for $k = 2$ to $|\boldsymbol{T}_{i,j}| - 1$ do // the first arc in a subtrip is always a dummy-arc//
    $u = T_{i,j,k}$;
    if $inv(u) \neq 0$ then
        Using $\boldsymbol{T}_{i,j}$, calculate $\Delta Z_{\text{flip}}$ depending on the relative position of $T_{i,j,k}$ as shown in
          Figure 5.3;
        if $\Delta Z_{\text{flip}} < \Delta Z^*$ then
            $\Delta Z^* = \Delta Z_{\text{flip}}$;
            $k^* = k$;
            // for first-move, the algorithm would stop here and immediately return the
              improving move's parameters // ;
return $(\Delta Z^*, k^*)$
```

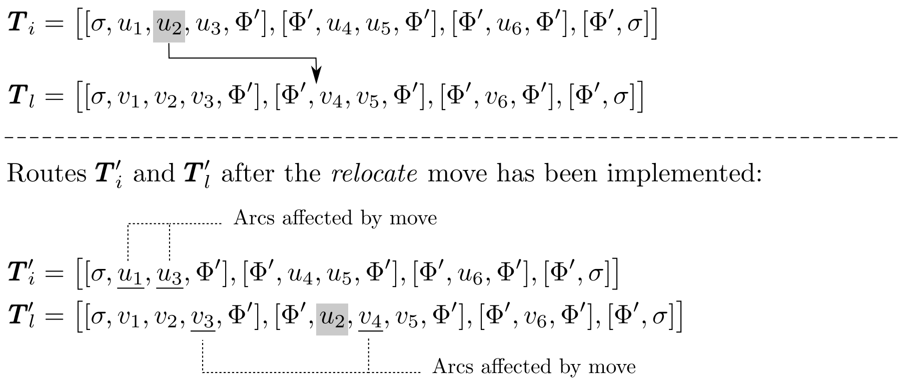
Algorithm 5.4: *Find-Flip-Move*

```
Input  : Current solution $\boldsymbol{T}$
Output: Cost-saving, $\Delta Z^*$; and the route, subtrip and suptrip positions, $i^*$, $j^*$, $k^*$, of the best
        move.
$\Delta Z^* = 0$;
$i^* = 0$; $j^* = 0$; $k^* = 0$;
for $i = 1$ to $|\boldsymbol{T}|$ do // the move is tested for each route //
    for $j = 1$ to $|\boldsymbol{T}_i| - 1$ do // the last subtrip is skipped since it contains only dummy arcs //
        $(\Delta Z^*_{\text{trip}}, k') =$ Find-Flip-Subtrip-Move$(\boldsymbol{T}_{i,j})$ // Algorithm 5.3 //;
        if $\Delta Z^*_{\text{trip}} < \Delta Z^*$ then
            $\Delta Z^* = \Delta Z^*_{\text{trip}}$;
            $i^* = i$; $j^* = j$; $k^* = k'$;
            // for first-move, the algorithm would stop here and immediately return the
              improving move's parameters //;
// if $k^* = 0$ no improving move could be found //;
return $(\Delta Z^*, i^*, j^*, k^*)$
```

5.3.2 Relocate

The second move operator that we addapt for the MCARPTIF is *relocate*, demonstrated in Figure 5.4. The operator relocates an arc 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 arc u. With a relocate position of $T_{l,m,n} = v$, arc 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 arc u after v,[2].

Original routes $\boldsymbol{T}_i$ and $\boldsymbol{T}_l$ with a planned move to *relocate* arc $T_{i,1,3} = u_2$ to position $T_{l,2,2} = v_4$:



$$\boldsymbol{T}_i = [[\sigma, u_1, \boxed{u_2}, u_3, \Phi'], [\Phi', u_4, u_5, \Phi'], [\Phi', u_6, \Phi'], [\Phi', \sigma]]$$

$$\boldsymbol{T}_l = [[\sigma, v_1, v_2, v_3, \Phi'], [\Phi', v_4, v_5, \Phi'], [\Phi', v_6, \Phi'], [\Phi', \sigma]]$$

Routes $\boldsymbol{T}'_i$ and $\boldsymbol{T}'_l$ after the *relocate* move has been implemented:

$$\boldsymbol{T}'_i = [[\sigma, \underline{u_1}, \underline{u_3}, \Phi'], [\Phi', u_4, u_5, \Phi'], [\Phi', u_6, \Phi'], [\Phi', \sigma]]$$

$$\boldsymbol{T}'_l = [[\sigma, v_1, v_2, \underline{v_3}, \Phi'], [\Phi', \boxed{u_2}, \underline{v_4}, v_5, \Phi'], [\Phi', v_6, \Phi'], [\Phi', \sigma]]$$

Figure 5.4: Example of the *relocate* move where an arc task is removed from a subtrip, and inserted into another position. The arc can be inserted in the same subtrip, a different subtrip in the same route, or in a subtrip of a different route.

The cost of the move consists of the cost of removing the arc from a subtrip, and the cost of inserting it into a new position. The removal cost depends on the position of

[2] The reason for our version inserting an arc before the arc in the target location is due to our algorithm implementation language, Python, and its *insert* function which works in this way.

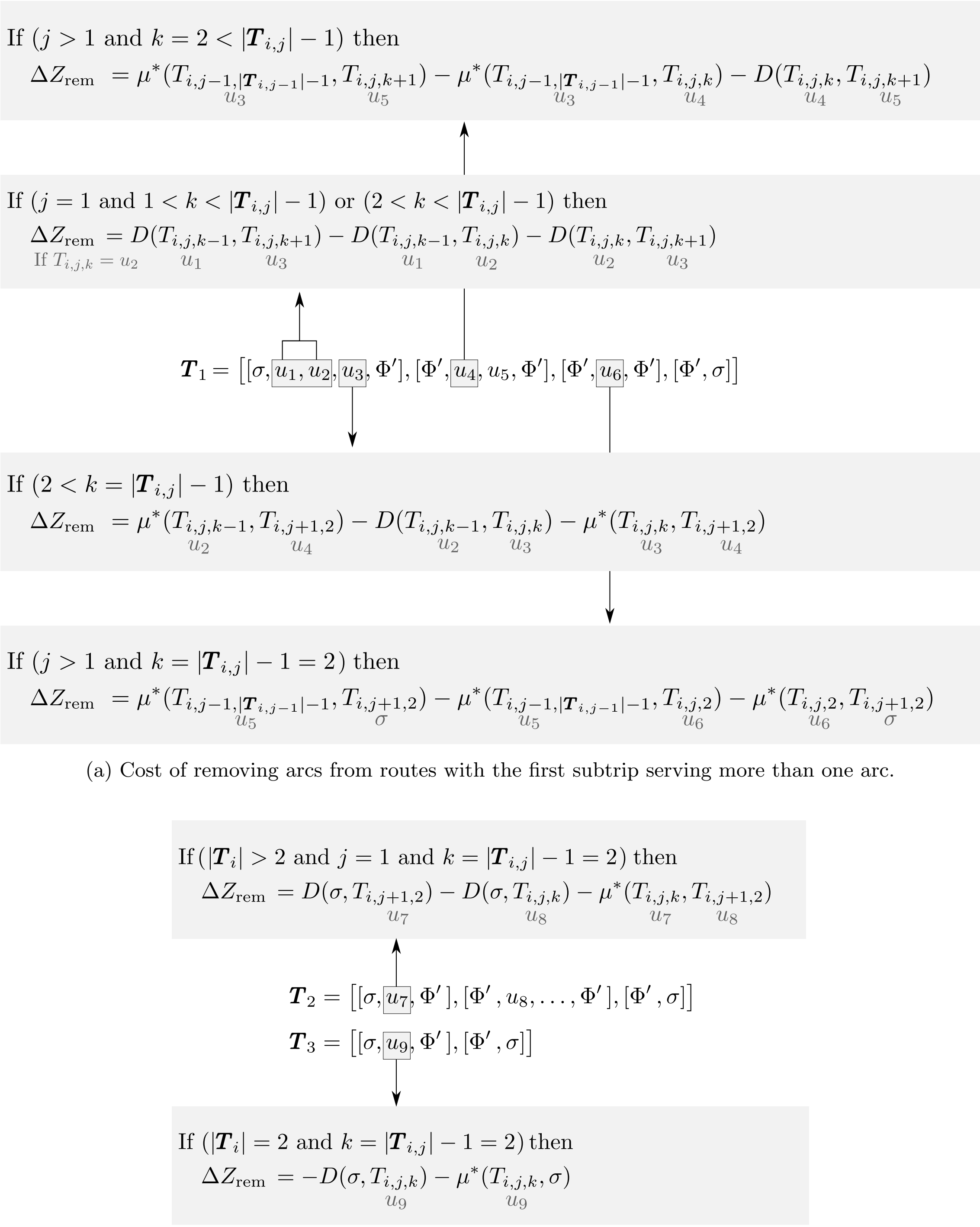
the arc relative to IF visits, with all possible cost formulas and the conditions for their applications shown in Figure 5.5. Included in Figure 5.5b is a condition for when an arc is removed from a route servicing only that arc, in which case the vehicle fleet size is reduced.

The cost of inserting an arc into a new position also depends on the insert position relative to IFs. The different insert-cost formulas are shown in Figure 5.6. The insert position may also be on an IF arc in the last position of a subtrip, in which case the arc is appended to the end of the subtrip, in-front of the IF arc.

Relocate may increase the demand and total cost of the route and subtrip to which arc u is relocated. As a result, feasibility checks have to be performed to determine if the move is feasible. The checks are illustrated in Figure 5.7. As a first condition, dummy arcs may not be removed from a subtrip, and an arc cannot be inserted in the first position of a substrip as it is reserved for depot or IF arcs. For an improving move, if the arc is relocated to the same subtrip, which we refer to as an *intra-subtrip* move, demand will remain unchanged and the cost of the route will be reduced. As a result, no feasibility checks on Q and L are required. The only condition is that the removed arc may not be reinserted into its current position or after itself. The same conditions are also checked when an arc is relocated to an adjacent subtrip. This condition prevents the last arc in a subtrip to be relocated to the first task in the adjacent subtrip, and vice-versa. This is equivalent to shifting the IF visit. Although such a move is valid, the optimal IF visits within a route can instead be determined using the IF splitting procedure for the MCARPIF. If an arc is relocated to a different subtrip in the same route, then a load capacity check is performed on the subtrip to which the arc is relocated. If it is relocated to a different route, then a load and route duration limit check is performed on the subtrip and route to which the arc is relocatd.

Algorithm 5.5 finds and returns the best *relocate* move between subtrips $\boldsymbol{T}_{i,j}$ and $\boldsymbol{T}_{l,m}$, where arcs are relocated from $\boldsymbol{T}_{i,j}$ to $\boldsymbol{T}_{l,m}$. The procedure also checks if inverting the arc before inserting it into $\boldsymbol{T}_{l,m}$ results in a better move. Algorithm 5.6 finds and returns the best *relocate* move among all subtrip pairs, including inter subtrip relocations. Each arc and its inverse is considered for relocation to the position before each other arc, as well as relocation to the end of subtrips. As such, the computational complexity of finding the best *relocate* move in an LS iterations is $\mathcal{O}(|\boldsymbol{R}| \times |\boldsymbol{R}_T|)$.

To improve the efficiency of finding the best *relocate* moves, and any improving move for that matter, the route costs and subtrip loads are stored prior to searching for the best improving move. The costs and loads are then updated, instead of recalculated, after a move is made. The sequence in which feasibility checks and cost calculations are performed can also be changed, depending on the characteristics of the problem instance under consideration. If the majority of moves are feasible, the move cost calculations can be performed first, and the feasibility checks will only have to be performed on improving moves. If all subtrips are close to capacity, and all routes are close to the route duration limit, then there may only be a few feasible moves available. In this case the load feasibility check can be performed first, followed by the move cost calculations, and lastly the route duration limit checks, since these depend on the move costs. We further applied two pre-move conditions to accelerate the search. Let $T_{i,j,k} = u$ be the arc to be relocated to a different subtrip $\boldsymbol{T}_{l,m}$, with $i \neq l$ or $i \neq m$. If the demand of u will result in $\boldsymbol{T}_{l,m}$ exceeding capacity, such that $load(\boldsymbol{T}_{l,m}) + q(u) > Q$, then all its insert positions will be infeasible in $\boldsymbol{T}_{l,m}$ and need not be evaluated. Similarly, let u be relocated to a different route $(i \neq l)$. If the service cost of u will result in $\boldsymbol{T}_l$ exceeding its time duration limit, such that $Z(\boldsymbol{T}_l) + w(u) > L$, then all its insertion positions in $\boldsymbol{T}_l$ will be infeasible and need not be evaluated.



If $(j > 1$ and $k = 2 < |\boldsymbol{T}_{i,j}| - 1)$ then

$$\Delta Z_{\text{rem}} = \mu^*(\underset{u_3}{T_{i,j-1,|\boldsymbol{T}_{i,j-1}|-1}}, \underset{u_5}{T_{i,j,k+1}}) - \mu^*(\underset{u_3}{T_{i,j-1,|\boldsymbol{T}_{i,j-1}|-1}}, \underset{u_4}{T_{i,j,k}}) - D(\underset{u_4}{T_{i,j,k}}, \underset{u_5}{T_{i,j,k+1}})$$

If $(j = 1$ and $1 < k < |\boldsymbol{T}_{i,j}| - 1)$ or $(2 < k < |\boldsymbol{T}_{i,j}| - 1)$ then

$$\underset{\text{If } T_{i,j,k} = u_2}{\Delta Z_{\text{rem}}} = D(\underset{u_1}{T_{i,j,k-1}}, \underset{u_3}{T_{i,j,k+1}}) - D(\underset{u_1}{T_{i,j,k-1}}, \underset{u_2}{T_{i,j,k}}) - D(\underset{u_2}{T_{i,j,k}}, \underset{u_3}{T_{i,j,k+1}})$$

$$\boldsymbol{T}_1 = \big[[\sigma, \boxed{u_1, u_2}, \boxed{u_3}, \Phi'], [\Phi', \boxed{u_4}, u_5, \Phi'], [\Phi', \boxed{u_6}, \Phi'], [\Phi', \sigma]\big]$$

If $(2 < k = |\boldsymbol{T}_{i,j}| - 1)$ then

$$\Delta Z_{\text{rem}} = \mu^*(\underset{u_2}{T_{i,j,k-1}}, \underset{u_4}{T_{i,j+1,2}}) - D(\underset{u_2}{T_{i,j,k-1}}, \underset{u_3}{T_{i,j,k}}) - \mu^*(\underset{u_3}{T_{i,j,k}}, \underset{u_4}{T_{i,j+1,2}})$$

If $(j > 1$ and $k = |\boldsymbol{T}_{i,j}| - 1 = 2)$ then

$$\Delta Z_{\text{rem}} = \mu^*(\underset{u_5}{T_{i,j-1,|\boldsymbol{T}_{i,j-1}|-1}}, \underset{\sigma}{T_{i,j+1,2}}) - \mu^*(\underset{u_5}{T_{i,j-1,|\boldsymbol{T}_{i,j-1}|-1}}, \underset{u_6}{T_{i,j,2}}) - \mu^*(\underset{u_6}{T_{i,j,2}}, \underset{\sigma}{T_{i,j+1,2}})$$

(a) Cost of removing arcs from routes with the first subtrip serving more than one arc.

If $(|\boldsymbol{T}_i| > 2$ and $j = 1$ and $k = |\boldsymbol{T}_{i,j}| - 1 = 2)$ then

$$\Delta Z_{\text{rem}} = D(\sigma, \underset{u_7}{T_{i,j+1,2}}) - D(\sigma, \underset{u_8}{T_{i,j,k}}) - \mu^*(\underset{u_7}{T_{i,j,k}}, \underset{u_8}{T_{i,j+1,2}})$$

$$\boldsymbol{T}_2 = \big[[\sigma, \boxed{u_7}, \Phi'], [\Phi', u_8, \ldots, \Phi'], [\Phi', \sigma]\big]$$

$$\boldsymbol{T}_3 = \big[[\sigma, \boxed{u_9}, \Phi'], [\Phi', \sigma]\big]$$

If $(|\boldsymbol{T}_i| = 2$ and $k = |\boldsymbol{T}_{i,j}| - 1 = 2)$ then

$$\Delta Z_{\text{rem}} = -D(\sigma, \underset{u_9}{T_{i,j,k}}) - \mu^*(\underset{u_9}{T_{i,j,k}}, \sigma)$$

(b) Cost of removing arcs from routes with the first subtrip serving only one arc.

Figure 5.5: Cost of removing arc u in position $T_{i,j,k}$ from a route. The cost depends on the removal position relative to the depot and intermediate facility visits.

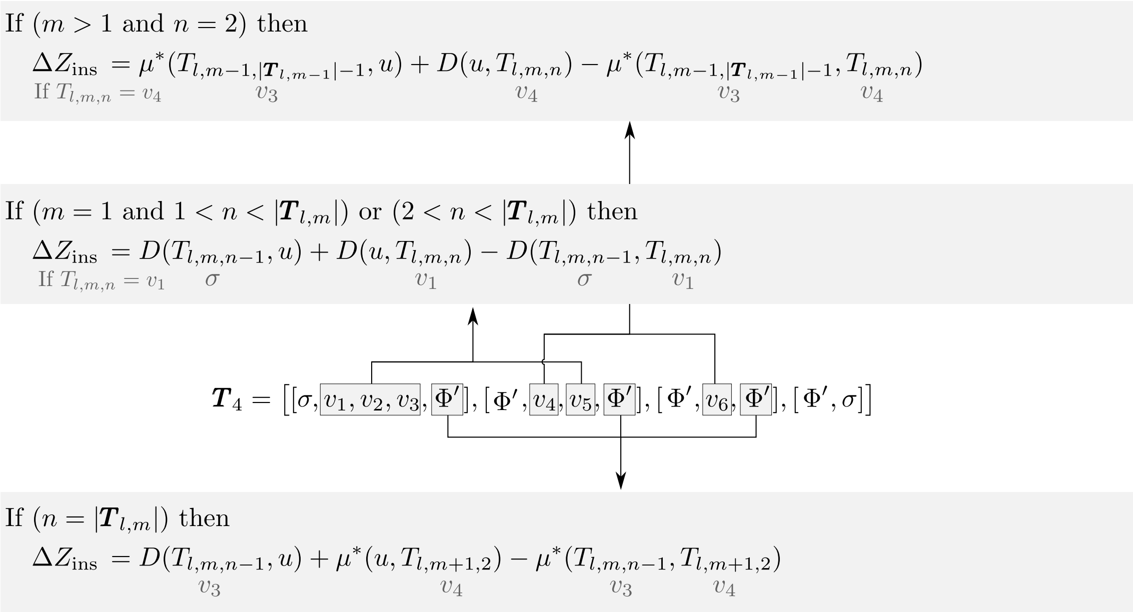

If ($m > 1$ and $n = 2$) then
$\Delta Z_{\text{ins}} = \mu^*(T_{l,m-1,|\boldsymbol{T}_{l,m-1}|-1}, u) + D(u, T_{l,m,n}) - \mu^*(T_{l,m-1,|\boldsymbol{T}_{l,m-1}|-1}, T_{l,m,n})$
If $T_{l,m,n} = v_4$ v_3 v_4 v_3 v_4
If ($m = 1$ and $1 < n < |\boldsymbol{T}_{l,m}|$) or ($2 < n < |\boldsymbol{T}_{l,m}|$) then
$\Delta Z_{\text{ins}} = D(T_{l,m,n-1}, u) + D(u, T_{l,m,n}) - D(T_{l,m,n-1}, T_{l,m,n})$
If $T_{l,m,n} = v_1$ σ v_1 σ v_1
$\boldsymbol{T}_4 = [[\sigma, v_1, v_2, v_3, \Phi'], [\Phi', v_4, v_5, \Phi'], [\Phi', v_6, \Phi'], [\Phi', \sigma]]$
If ($n = |\boldsymbol{T}_{l,m}|$) then
$\Delta Z_{\text{ins}} = D(T_{l,m,n-1}, u) + \mu^*(u, T_{l,m+1,2}) - \mu^*(T_{l,m,n-1}, T_{l,m+1,2})$
v_3 v_4 v_3 v_4

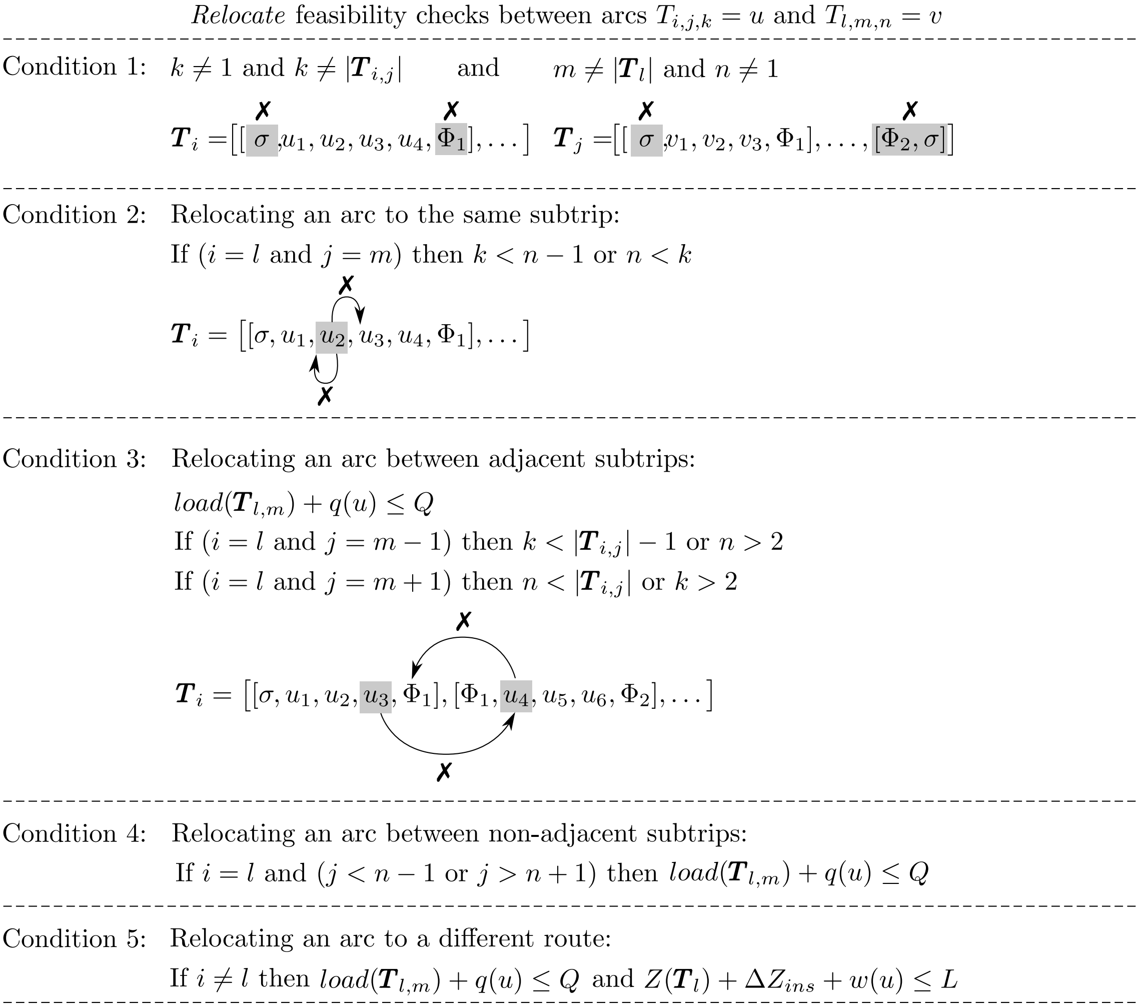
Figure 5.6: Cost of inserting arc u in position $T_{l,m,n}$. The costs depends on the insert position relative to intermediate facility visits.

Algorithm 5.5: *Find-Relocate-Subtrip-Move*

Input : Subtrips $\boldsymbol{T}_{i,j}$ from which arcs are to be removed, and $\boldsymbol{T}_{l,m}$ into which the removed arcs are to be relocated.

Output: Cost-saving, ΔZ^*; arc positions, k^* and n^*; and arc orientation, u^*, of the best subtrip move.

```
1  $\Delta Z^* = 0$;
2  $k^* = 0$; $n^* = 0$; $u^* = 0$;
3  for $k = 2$ to $|\boldsymbol{T}_{i,j}| - 1$ do // only arc tasks can be relocated //
4      $u = T_{i,j,k}$;
5      Using $\boldsymbol{T}_{i,j}$, calculate $\Delta Z_{\text{rem}}$ depending on the relative position of $T_{i,j,k}$ as shown in Figure 5.5;
6      for $n = 2$ to $|\boldsymbol{T}_{l,m}|$ do // an arc can be inserted on the last arc in a subtrip //
7          Using $u$ and $\boldsymbol{T}_{l,m}$, calculate $\Delta Z_{\text{in}}$ depending on the relative position of $T_{l,m,n}$ as shown in
             Figure 5.6;
8          $\Delta Z = \Delta Z_{\text{rem}} + \Delta Z_{\text{in}}$;
9          if all *relocate* conditions shown in Figure 5.7 are met then
10             if $\Delta Z < \Delta Z^*$ then
11                 $\Delta Z^* = \Delta Z$;
12                 $k^* = k$; $n^* = n$; $u^* = u$;
13                 // for *first-move*, the algorithm would stop here and immediately return the
                     improving move's parameters //;
14     if $inv(u) \neq 0$ then // inverting and then inserting the removed arc is also evaluated //
15         Set $u = inv(u)$;
16         Repeat lines 6–12;
17         Reset $u = T_{i,j,k}$;
18 return ($\Delta Z^*$, $k^*$, $n^*$, $u^*$)
```



Relocate feasibility checks between arcs $T_{i,j,k} = u$ and $T_{l,m,n} = v$

Condition 1: $k \neq 1$ and $k \neq |\boldsymbol{T}_{i,j}|$ and $m \neq |\boldsymbol{T}_l|$ and $n \neq 1$

$\boldsymbol{T}_i = [[\sigma, u_1, u_2, u_3, u_4, \Phi_1], \ldots]$ $\boldsymbol{T}_j = [[\sigma, v_1, v_2, v_3, \Phi_1], \ldots, [\Phi_2, \sigma]]$

Condition 2: Relocating an arc to the same subtrip:

If $(i = l$ and $j = m)$ then $k < n - 1$ or $n < k$

$\boldsymbol{T}_i = [[\sigma, u_1, u_2, u_3, u_4, \Phi_1], \ldots]$

Condition 3: Relocating an arc between adjacent subtrips:

$load(\boldsymbol{T}_{l,m}) + q(u) \leq Q$

If $(i = l$ and $j = m - 1)$ then $k < |\boldsymbol{T}_{i,j}| - 1$ or $n > 2$

If $(i = l$ and $j = m + 1)$ then $n < |\boldsymbol{T}_{i,j}|$ or $k > 2$

$\boldsymbol{T}_i = [[\sigma, u_1, u_2, u_3, \Phi_1], [\Phi_1, u_4, u_5, u_6, \Phi_2], \ldots]$

Condition 4: Relocating an arc between non-adjacent subtrips:

If $i = l$ and $(j < n - 1$ or $j > n + 1)$ then $load(\boldsymbol{T}_{l,m}) + q(u) \leq Q$

Condition 5: Relocating an arc to a different route:

If $i \neq l$ then $load(\boldsymbol{T}_{l,m}) + q(u) \leq Q$ and $Z(\boldsymbol{T}_l) + \Delta Z_{ins} + w(u) \leq L$

Figure 5.7: Conditions for *relocating* arc $T_{i,j,k}$ to position $T_{l,m,n}$.

Algorithm 5.6: *Find-Relocate-Move*

```
Input  : Current solution $\boldsymbol{T}$.
Output: Cost-saving, $\Delta Z^*$; route, subtrip, and arc positions, $i^*$, $j^*$, $k^*$, $l^*$, $m^*$, $n^*$; and arc
        orientation, $u^*$, of the best move.
$\Delta Z^* = 0$;
$i^* = 0$; $j^* = 0$; $k^* = 0$;
$l^* = 0$; $m^* = 0$; $n^* = 0$;
$u^* = 0$;
for $i = 1$ to $|\boldsymbol{T}|$ do
    for $j = 1$ to $|\boldsymbol{T}_i| - 1$ do
        for $l = 1$ to $|\boldsymbol{T}|$ do
            for $m = 1$ to $|\boldsymbol{T}_l| - 1$ do
                $(\Delta Z^*_{\text{trips}}, k', n', u') =$ Find-Relocate-Subtrip-Move$(\boldsymbol{T}_{i,j}, \boldsymbol{T}_{l,m})$ // Algorithm 5.5 //;
                if $\Delta Z^*_{trips} < \Delta Z^*$ then
                    $\Delta Z^* = \Delta Z^*_{\text{trips}}$;
                    $i^* = i$; $j^* = j$; $k^* = k'$;
                    $l^* = l$; $m^* = m$; $n^* = n'$;
                    $u^* = u'$;
                    // for first-move, the algorithm would stop here and immediately return the
                      improving move's parameters //;
// if $k^* = 0$ no improving move could be found //;
return $(\Delta Z^*, i^*, j^*, k^*, l^*, m^*, n^*, u^*)$
```

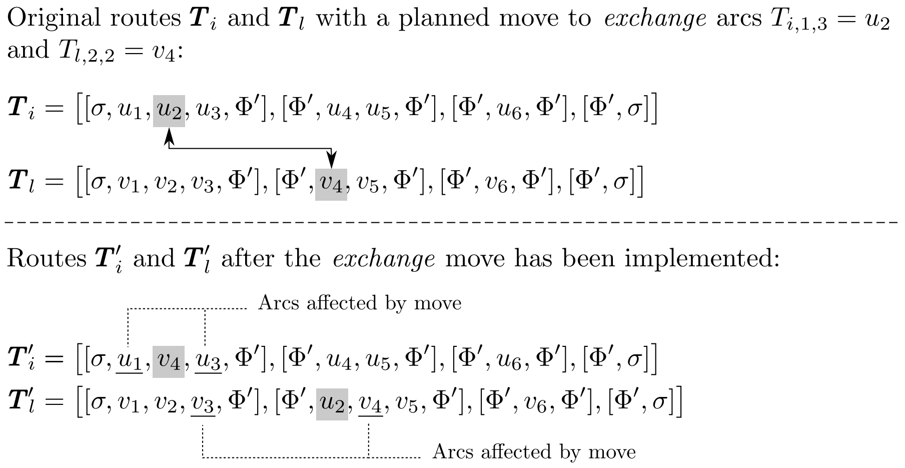
5.3.3 Exchange

The third move operator that we adapt for the MCARPTIF is *exchange*, illustrated in Figure 5.8. An *exchange* move involving $T_{i,j,k} = u$ and $T_{l,m,n} = v$ consists of replacing arc u with v, and v with u. The cost of the move is then the sum of the two replacement costs. The cost of replacing an arc, u, with another arc, v, can be calculated as shown in Figure 5.9, which depends on the position of u relative to IF visits.

The conditions for an *exchange* move to be feasible are shown in Figure 5.10. The depot and IF arcs cannot be exchanged. When arcs are in the same subtrip, an arc cannot be exchanged with itself, and not with its preceding or following arc. For the latter, the move is identical to *relocating* an arc in-front of its predecessor. The cost of such a move is not covered by the replacement costs shown in Figure 5.9 and is thus not evaluated with *exchange*. When arcs u and v are in different subtrips, the load changes of both substrips have to be checked for capacity violations. Similarly, if the arcs are in different routes, the route cost changes also have to be checked against route duration limits.

Algorithm 5.7 finds and returns the best *exchange* move between subtrips $\boldsymbol{T}_{i,j}$ and $\boldsymbol{T}_{l,m}$. Exchanging arc u with arc v is the same is exchanging arc v with arc u, hence why line 5 is specified for $n = k$ to $|\boldsymbol{T}_{l,m}| - 1$. This avoids the same move being evaluated twice. The algorithm also checks if using either one or both of the inverted arcs as a replacement results in a better move. If both u and v can be inverted, there are four exchange combinations that are checked between u and v.

Algorithm 5.8 finds and returns the best *exchange* move among all subtrip pairs, including exchanging arcs within the same subtrip. Each two arc combination set, $\{u, v\} \in \boldsymbol{R}_T$, is considered for exchange. Since inverting the arcs is also considered the computational complexity of finding the best *exchange* move in an LS iteration is $\mathcal{O}(\boldsymbol{R}^2)$. The load and route cost changes are unique per move, so we do not apply pre-move feasibility checks to accelerate the search.



Original routes $\boldsymbol{T}_i$ and $\boldsymbol{T}_l$ with a planned move to *exchange* arcs $T_{i,1,3} = u_2$ and $T_{l,2,2} = v_4$:

$$\boldsymbol{T}_i = \big[[\sigma, u_1, u_2, u_3, \Phi'], [\Phi', u_4, u_5, \Phi'], [\Phi', u_6, \Phi'], [\Phi', \sigma]\big]$$

$$\boldsymbol{T}_l = \big[[\sigma, v_1, v_2, v_3, \Phi'], [\Phi', v_4, v_5, \Phi'], [\Phi', v_6, \Phi'], [\Phi', \sigma]\big]$$

Routes $\boldsymbol{T}'_i$ and $\boldsymbol{T}'_l$ after the *exchange* move has been implemented:

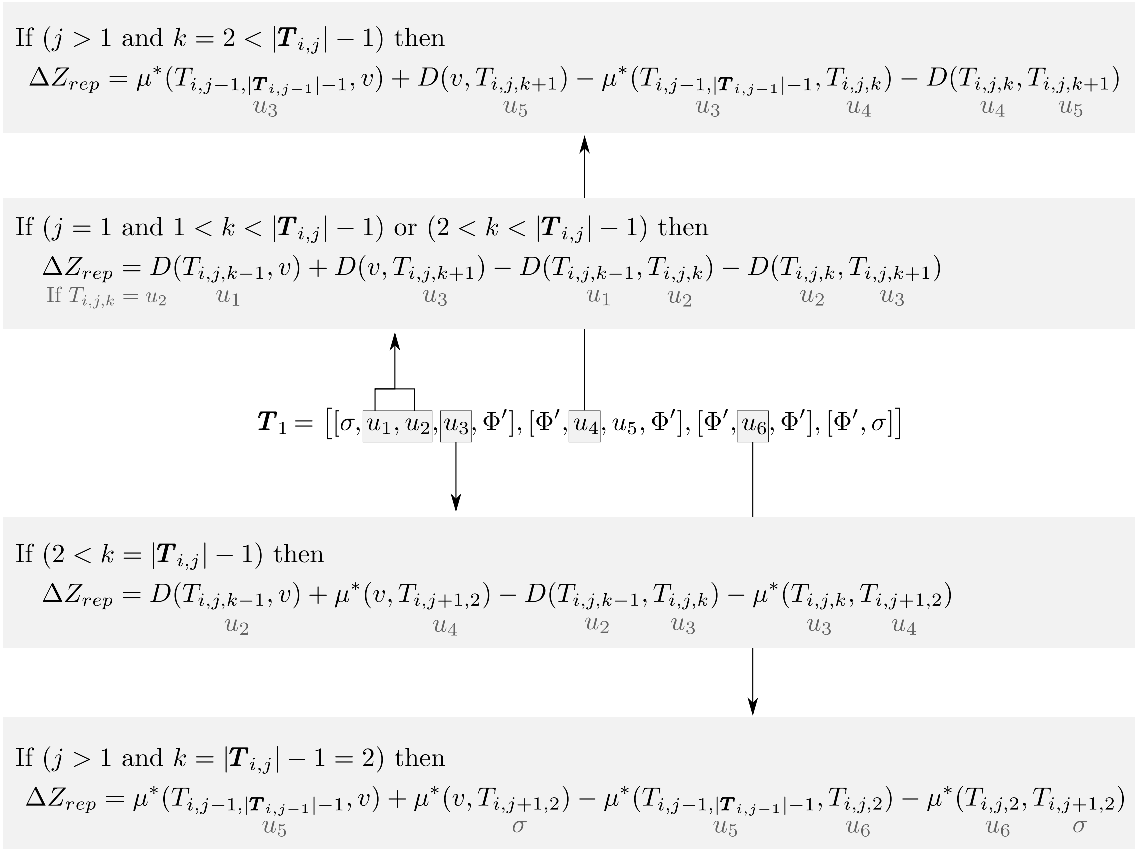
Arcs affected by move

$$\boldsymbol{T}'_i = \big[[\sigma, \underline{u_1}, v_4, \underline{u_3}, \Phi'], [\Phi', u_4, u_5, \Phi'], [\Phi', u_6, \Phi'], [\Phi', \sigma]\big]$$

$$\boldsymbol{T}'_l = \big[[\sigma, v_1, v_2, \underline{v_3}, \Phi'], [\Phi', u_2, \underline{v_4}, v_5, \Phi'], [\Phi', v_6, \Phi'], [\Phi', \sigma]\big]$$

Arcs affected by move

Figure 5.8: Example of the *exchange* move where the position of two arc tasks are swapped. The tasks can be in the same subtrip, different subtrips in the same route, or in subtrips of different routes.



If $(j > 1$ and $k = 2 < |\boldsymbol{T}_{i,j}| - 1)$ then

$$\Delta Z_{rep} = \mu^*(\underset{u_3}{T_{i,j-1,|\boldsymbol{T}_{i,j-1}|-1}}, v) + D(v, \underset{u_5}{T_{i,j,k+1}}) - \mu^*(\underset{u_3}{T_{i,j-1,|\boldsymbol{T}_{i,j-1}|-1}}, \underset{u_4}{T_{i,j,k}}) - D(\underset{u_4}{T_{i,j,k}}, \underset{u_5}{T_{i,j,k+1}})$$

If $(j = 1$ and $1 < k < |\boldsymbol{T}_{i,j}| - 1)$ or $(2 < k < |\boldsymbol{T}_{i,j}| - 1)$ then

$$\Delta Z_{rep} = D(\underset{u_1}{T_{i,j,k-1}}, v) + D(v, \underset{u_3}{T_{i,j,k+1}}) - D(\underset{u_1}{T_{i,j,k-1}}, \underset{u_2}{T_{i,j,k}}) - D(\underset{u_2}{T_{i,j,k}}, \underset{u_3}{T_{i,j,k+1}})$$

If $T_{i,j,k} = u_2$

$$\boldsymbol{T}_1 = \big[[\sigma, u_1, u_2, u_3, \Phi'], [\Phi', u_4, u_5, \Phi'], [\Phi', u_6, \Phi'], [\Phi', \sigma]\big]$$

If $(2 < k = |\boldsymbol{T}_{i,j}| - 1)$ then

$$\Delta Z_{rep} = D(\underset{u_2}{T_{i,j,k-1}}, v) + \mu^*(v, \underset{u_4}{T_{i,j+1,2}}) - D(\underset{u_2}{T_{i,j,k-1}}, \underset{u_3}{T_{i,j,k}}) - \mu^*(\underset{u_3}{T_{i,j,k}}, \underset{u_4}{T_{i,j+1,2}})$$

If $(j > 1$ and $k = |\boldsymbol{T}_{i,j}| - 1 = 2)$ then

$$\Delta Z_{rep} = \mu^*(\underset{u_5}{T_{i,j-1,|\boldsymbol{T}_{i,j-1}|-1}}, v) + \mu^*(v, \underset{\sigma}{T_{i,j+1,2}}) - \mu^*(\underset{u_5}{T_{i,j-1,|\boldsymbol{T}_{i,j-1}|-1}}, \underset{u_6}{T_{i,j,2}}) - \mu^*(\underset{u_6}{T_{i,j,2}}, \underset{\sigma}{T_{i,j+1,2}})$$

Figure 5.9: Cost of replacing arc $T_{i,j,k} = u$ with arc v. The cost depends on the replacement arc's position relative to intermediate facility visits.

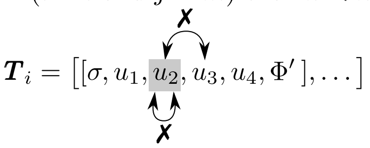
Exchange feasibility check between arcs $T_{i,j,k} = u$ and $T_{l,m,n} = v$

Condition 1: $k \neq 1$ and $k \neq |\boldsymbol{T}_{i,j}|$ and $n \neq 1$ and $n \neq |\boldsymbol{T}_{l,m}|$

$$\boldsymbol{T}_i = [[\overset{\text{✗}}{\boxed{\sigma}}, u_1, u_2, u_3, u_4, \overset{\text{✗}}{\boxed{\Phi'}}], \ldots] \quad \boldsymbol{T}_j = [[\overset{\text{✗}}{\boxed{\sigma}}, v_1, v_2, v_3, \overset{\text{✗}}{\boxed{\Phi'}}], \ldots]$$

Condition 2: Exchanging arcs in the same suptrip:

If $(i = l$ and $j = m)$ then $k < n - 1$ or $n < k$

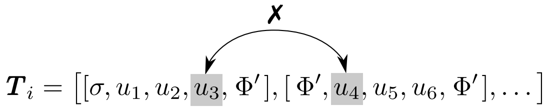

✗
$\boldsymbol{T}_i = [[\sigma, u_1, u_2, u_3, u_4, \Phi'], \ldots]$
✗

Condition 3: Exchanging arcs in adjacent suptrips:

$load(\boldsymbol{T}_{i,j}) - q(u) + q(w) \leq Q$ and $load(\boldsymbol{T}_{l,m}) - q(v) + q(u) \leq Q$

If $(i = l$ and $j = m - 1)$ then $k < |\boldsymbol{T}_{i,j}| - 1$ or $n > 2$

If $(i = l$ and $j = m + 1)$ then $n < |\boldsymbol{T}_{i,j}| - 1$ or $k > 2$


✗
$\boldsymbol{T}_i = [[\sigma, u_1, u_2, u_3, \Phi'], [\Phi', u_4, u_5, u_6, \Phi'], \ldots]$

Condition 4: Exchanging arcs in non-adjacent subtrips:

If $i = l$ and $(j < n - 1$ or $j > n + 1)$ then

$load(\boldsymbol{T}_{i,j}) - q(u) + q(v) \leq Q$ and $load(\boldsymbol{T}_{l,m}) - q(v) + q(u) \leq Q$

Condition 5: Exchanging arcs in different routes:

$load(\boldsymbol{T}_{i,j}) - q(u) + q(v) \leq Q$ and $load(\boldsymbol{T}_{l,m}) - q(v) + q(u) \leq Q$

and $Z(\boldsymbol{T}_i) + \Delta Z_{rep_u} - w(u) + w(v) \leq L$

and $Z(\boldsymbol{T}_l) + \Delta Z_{rep_v} - w(v) + w(u) \leq L$

Figure 5.10: Conditions for *exchanging* arcs $T_{i,j,k}$ and $T_{l,m,n}$.

Algorithm 5.7: *Find-Exchange-Subtrip-Move*

Input : Subtrips $\boldsymbol{T}_{i,j}$ and $\boldsymbol{T}_{l,m}$ between which the move is applied.

Output: Cost-saving, ΔZ^*; arc positions, k^* and n^*; and arc orientations, u^* and v^*, of the best suptrip move.

```
1  $\Delta Z^* = 0$;
2  $k^* = 0$; $n^* = 0$; $u^* = 0$;
3  for $k = 2$ to $|\boldsymbol{T}_{i,j}| - 2$ do
4      $u = T_{i,j,k}$;
5      for $n = k$ to $|\boldsymbol{T}_{l,m}| - 1$ do // exchanging $\{u, v\}$ is the same as $\{v, u\}$ //
6          $v = T_{l,m,n}$;
7          Using $v$ and $\boldsymbol{T}_{i,j}$, calculate $\Delta Z_{\text{rep}_u}$ depending on the relative position of $T_{i,j,k}$ as shown in
             Figure 5.9;
8          Using $u$ and $\boldsymbol{T}_{l,m}$, calculate $\Delta Z_{\text{rep}_v}$ depending on the relative position of $T_{l,m,n}$ as shown
             in Figure 5.9;
9          if all exchange conditions shown in Figure 5.10 are met then
10             $\Delta Z = \Delta Z_{\text{rep}_u} + \Delta Z_{\text{rep}_v}$;
11             if $\Delta Z < \Delta Z^*$ then
12                 $\Delta Z^* = \Delta Z$;
13                 $k^* = k$; $n^* = n$; $u^* = u$; $v^* = v$;
14                 // for first-move, the algorithm would stop here and immediately return the
                     improving move's parameters //;
15         if $inv(u) \neq 0$ then
16             Set $u = inv(u)$;
17             Repeat lines 7–13;
18             Reset $u = T_{i,j,k}$;
19         if $inv(v) \neq 0$ then
20             Set $v = inv(v)$;
21             Repeat lines 7–13;
22             Reset $v = T_{l,m,n}$;
23         if $inv(u) \neq 0$ and $inv(v) \neq 0$ then
24             Set $u = inv(u)$; $v = inv(v)$;
25             Repeat lines 7–13;
26             Reset $u = T_{i,j,k}$; $v = T_{l,m,n}$;
27 return ($\Delta Z^*$, $k^*$, $n^*$, $u^*$, $v^*$)
```

Algorithm 5.8: *Find-Exchange-Move*

```
Input  : Current solution $\boldsymbol{T}$.
Output: Cost-saving, $\Delta Z^*$; route, subtrip, and arc positions, $i^*$, $j^*$, $k^*$, $l^*$, $m^*$, $n^*$; and arc
        orientations, $u^*$ and $v^*$, of the best move.
$\Delta Z^* = 0$;
$i^* = 0$; $j^* = 0$; $k^* = 0$;
$l^* = 0$; $m^* = 0$; $n^* = 0$;
$u^* = 0$;
$v^* = 0$;
for $i = 1$ to $|\boldsymbol{T}| - 1$ do
    for $j = 1$ to $|\boldsymbol{T}_i| - 1$ do
        for $l = i$ to $|\boldsymbol{T}|$ do
            if $i = l$ then $j' = j$;
            else $j = 1$;
            for $m = j'$ to $|\boldsymbol{T}_l| - 1$ do
                $(\Delta Z^*_{trips}, k', n', u', v') =$ Find-Exchange-Subtrip-Move$(\boldsymbol{T}_{i,j}, \boldsymbol{T}_{l,m})$ //
                  Algorithm 5.7 //;
                if $\Delta Z^*_{\text{trips}} < \Delta Z^*$ then
                    $\Delta Z^* = \Delta Z^*_{\text{trips}}$;
                    $i^* = i$; $j^* = j$; $k^* = k'$;
                    $l^* = l$; $m^* = m$; $n^* = n'$;
                    $u^* = u'$; $v^* = v'$;
                    // for first-move, the algorithm would stop here and immediately return the
                      improving move's parameters //;
// if $k^* = 0$ no improving move could be found //;
return $(\Delta Z^*, i^*, j^*, k^*, l^*, m^*, n^*, u^*, v^*)$
```

5.3.4 Cross

The second last operator that we develop for the MCARPTIF is *cross*, which is a version of *two-opt* performed between two different routes. The *cross* move is illustrated in Figure 5.11. First, the two different routes are split at specific positions to form four partial routes. The two partial routes representing the end portions of the original routes are then swapped and linked with the two partial routes representing the beginning portions of the original routes. Routes are crossed at arcs $T_{i,j,k} = u$ and $T_{l,m,n} = v$ with splits performed between $T_{i,j,k-1}$ and u, and between $T_{l,m,n-1}$ and v. The cost of the move consists of the cost of splitting the route, and then linking the resulting partial routes.

The most regular *cross* move involves positions that are preceded by arc tasks, with the conditions for the move, as well as its cost and feasibility checks shown in Figure 5.12. To check the feasibility of the move it is necessary to determine subtrip loads as well as route costs of all four partial routes, resulting from the two splits. The combined load of subtrips from partial routes are then checked against vehicle capacity limits, and the cost of the combined routes are checked against the route duration limit.

The move costs and feasibility checks for cases where $T_{i,j,k} = u$ is preceded by an arc task, and $T_{l,m,n} = v$ is either the last or first IF, or the first arc after an IF, are shown in Section 5.B at the end of the chapter. In certain cases, partial routes are linked through IF visits and the load feasibility check need not be performed. In other cases, the move reduces the total number of IF visits over both routes. There are also unique moves between arcs that are not preceded by arc tasks, including a move that results in an entire route being appended to another, thus reducing the fleet size. A number of moves results in the same neighbouring solution, and all need not be evaluated. A few moves

Original routes with a planned move to *cross* the routes at arcs $T_{i,1,4} = u_3$ to position $T_{l,2,4} = v_3$:

$$\boldsymbol{T}_i = [[\sigma, u_1, u_2, u_3, \Phi'], [\Phi', u_4, u_5, \Phi'], [\Phi', \sigma]]$$
$$\boldsymbol{T}_l = [[\sigma, v_1, v_2, v_3, \Phi'], [\Phi', v_4, v_5, \Phi'], [\Phi', \sigma]]$$

Step 1: The routes are *split* at arcs u_3 and v_3 to form four temporary incomplete-routes:

$$\boldsymbol{T}_i = [[\sigma, u_1, u_2, \longrightarrow u_3, \Phi'], [\Phi', u_4, u_5, \Phi'], [\Phi', \sigma]]$$
$$\boldsymbol{T}_l = [[\sigma, v_1, v_2, \longrightarrow v_3, \Phi'], [\Phi', v_4, v_5, \Phi'], [\Phi', \sigma]]$$

$$\boldsymbol{T}'_{i_a} = [[\sigma, u_1, u_2]] \qquad \boldsymbol{T}'_{i_b} = [[u_3, \Phi'], [\Phi', u_4, u_5, \Phi'] \ [\Phi', \sigma]]$$
$$\boldsymbol{T}'_{l_a} = [[\sigma, v_1, v_2]] \qquad \boldsymbol{T}'_{l_b} = [[v_3, \Phi'], [\Phi', v_4, v_5, \Phi'], [\Phi', \sigma]]$$

Step 2: Temporary route $\boldsymbol{T}'_{i_a}$ is *linked* with $\boldsymbol{T}'_{l_b}$, and $\boldsymbol{T}'_{l_a}$ with $\boldsymbol{T}'_{i_b}$ to form two new complete-routes:

$$\begin{tikzcd}
\boldsymbol{T}'_{i_a} = [[\sigma, u_1, u_2]] \arrow[dr] & \boldsymbol{T}'_{i_b} = [[u_3, \Phi'], [\Phi', u_4, u_5, \Phi'], [\Phi', \sigma]] \\
\boldsymbol{T}'_{l_a} = [[\sigma, v_1, v_2]] \arrow[ur] & \boldsymbol{T}'_{l_b} = [[v_3, \Phi'], [\Phi', v_4, v_5, \Phi'], [\Phi', \sigma]]
\end{tikzcd}$$

Routes $\boldsymbol{T}'_i$ and $\boldsymbol{T}'_l$ after the *cross* move has been implemented:

Arcs affected by move

$$\boldsymbol{T}'_i = [[\sigma, u_1, \underline{u_2}, v_3, \Phi'], [\Phi', v_4, v_5, \Phi'], [\Phi', \sigma]]$$
$$\boldsymbol{T}'_l = [[\sigma, v_1, \underline{v_2}, u_3, \Phi'], [\Phi', u_4, u_5, \Phi'], [\Phi', \sigma]]$$

Arcs affected by move

Figure 5.11: Example of the *cross* move where the end portions of two different routes are swapped at the positions of arcs $T_{i,j,k}$ and $T_{l,m,n}$.

If $k < |\boldsymbol{T}_{i,j}|$ and $n < |\boldsymbol{T}_{l,m}|$ then:

$\boldsymbol{T}_1 = [[\sigma, u_1, u_2, u_3, \Phi'], [\Phi', u_4, u_5, \Phi'], [\Phi', \sigma]]$ $\quad$ $\boldsymbol{T}_2 = [[\sigma, v_1, v_2, v_3, \Phi'], [\Phi', v_4, v_5, \Phi'], [\Phi', \sigma]]$

Routes after *cross* move: $\boldsymbol{T}'_1 = [[\sigma, u_1, u_2, \underline{v_3}, \Phi'], [\Phi', v_4, v_5, \Phi'], [\Phi', \sigma]]$ $\boldsymbol{T}'_2 = [[\sigma, v_1, v_2, \underline{u_3}, \Phi'], [\Phi', u_4, u_5, \Phi'], [\Phi', \sigma]]$

$\boldsymbol{T}'_{1_a} = [[\sigma, u_1, u_2]]$ $\boldsymbol{T}'_{1_b} = [[\underline{u_3}, \Phi'], [\Phi', u_4, u_5, \Phi'], [\Phi', \sigma]]$ $\quad$ $\boldsymbol{T}'_{2_a} = [[\sigma, v_1, v_2]]$ $\boldsymbol{T}'_{2_b} = [[\underline{v_3}, \Phi'], [\Phi', v_4, v_5, \Phi'], [\Phi', \sigma]]$

Move costs

$\Delta Z_{split_i} = D(\underset{u_2}{T_{i,j,k-1}}, \underset{u_3}{T_{i,j,k}})$ $\quad$ $\Delta Z_{split_l} = D(\underset{v_2}{T_{l,m,n-1}}, \underset{v_3}{T_{l,m,n}})$

$\Delta Z_{link_i} = D(\underset{u_2}{T_{i,j,k-1}}, \underset{v_3}{T_{l,m,n}})$ $\quad$ $\Delta Z_{link_l} = D(\underset{v_2}{T_{l,m,n-1}}, \underset{u_3}{T_{i,j,k}})$

$\Delta Z = Z_{link_i} - Z_{split_i} + Z_{link_l} - Z_{split_l}$

Partial-routes

$\boldsymbol{T}'_{i_a} = [\ldots, [\ldots, T_{i,j,k-1}]]$ $\boldsymbol{T}'_{i_b} = [[T_{i,j,k}, \ldots], \ldots]$ $\quad$ $\boldsymbol{T}'_{l_a} = [\ldots, [\ldots, T_{l,m,n-1}]]$ $\boldsymbol{T}'_{l_b} = [[T_{l,m,n}, \ldots], \ldots]$

Feasibility check:

$load(\boldsymbol{T}'_{i_a,j}) + load(\boldsymbol{T}'_{l_b,1}) \leq Q$ and $\quad$ $load(\boldsymbol{T}'_{l_a,m}) + load(\boldsymbol{T}'_{i_b,1}) \leq Q$ and

$Z(\boldsymbol{T}'_{i_a}) + \Delta Z_{link_i} + Z(\boldsymbol{T}'_{l_b}) \leq L$ and $\quad$ $Z(\boldsymbol{T}'_{l_a}) + \Delta Z_{link_l} + Z(\boldsymbol{T}'_{i_b}) \leq L$

Figure 5.12: The cost and feasibility checks when two routes are *crossed* at positions preceded by arc tasks.

also result in moves identical to *relocate* and *exchange* moves, particularly when a *cross* move involves the last arc tasks of both routes. Such moves can be skipped if *relocate* and *exchange* are used in conjunction with *cross*.

The best *cross* move between subtrips $\boldsymbol{T}_{i,j}$ and $\boldsymbol{T}_{l,m}$, where $i \neq l$ can be found using Algorithm 5.9, and the best *cross* move among all subtrip pairs belonging to different routes can be found using Algorithm 5.10. Each unique two arc combination between different routes is considered for a *cross*. The computational complexity of finding the best *cross* move is thus $\mathcal{O}(|\boldsymbol{R}_T|^2)$.

Similar to other move operators, route costs and subtrip loads can be stored prior to searching for the best improving move and updated after each move. For the feasibility checks, the load of subtrips in partial routes, as well as the cost of partial routes have to calculated, which can be done using the following representation. For load calculations, the accumulated load on each position in a subtrip is stored in a list $\mathscr{L}$. Let $\mathscr{L}_{i,j,k}$ be the accumulated load of subtrip $\boldsymbol{T}_{i,j}$ at position k, which is equal to $\mathscr{L}_{i,j,k-1} + q(T_{i,j,k})$ where $k > 1$ and $\mathscr{L}_{i,j,1} = 0$. Using the same notation as Figure 5.12, if a *cross* move is to be performed on $T_{i,j,k} = u$, then the loads of the last and first subtrips, $\boldsymbol{T}'_{i_a,j}$ and $\boldsymbol{T}'_{i_b,j}$, of the partial routes are given by

$$load(\boldsymbol{T}'_{i_a,j}) = \mathscr{L}_{i,j,k-1}, \tag{5.1}$$

$$load(\boldsymbol{T}'_{i_b,j}) = load(\boldsymbol{T}_{i,j}) - \mathscr{L}_{i,j,k-1}, \text{ or} \tag{5.2}$$

$$load(\boldsymbol{T}'_{i_b,j}) = \mathscr{L}_{i,j,|\boldsymbol{T}_{i,j}|} - \mathscr{L}_{i,j,k}. \tag{5.3}$$

Using Equations (5.1) to (5.3) the load feasibility of a move can be efficiently determined. The costs of the partial routes can be calculated in a similar way. Let $\mathscr{Z}_{i,j,k}$ be the cost

Algorithm 5.9: *Find-Cross-Subtrip-Move*

```
Input  : Subtrips $\boldsymbol{T}_{i,j}$ and $\boldsymbol{T}_{l,m}$ between which the move is applied.
Output: Cost-saving, $\Delta Z^*$; and arc positions, $k^*$ and $n^*$, of the best suptrip move.
$\Delta Z^* = 0$;
$k^* = 0$; $n^* = 0$;
for $k = 1$ to $|\boldsymbol{T}_{i,j}| - 1$ do
    for $n = k$ to $|\boldsymbol{T}_{l,m}|$ do
        Using $T_{i,j,k}$ and $T_{l,m,n}$, calculate $\Delta Z$ depending on the relative positions of $T_{i,j,k}$ and
          $T_{l,m,n}$ as shown in Figures 5.12 and 5.19 to 5.22;
        if all cross conditions shown in Figures 5.12 and 5.19 to 5.22 are met then
            if $\Delta Z < \Delta Z^*$ then
                $\Delta Z^* = \Delta Z$;
                $k^* = k$; $n^* = n$;
                // for first-move, the algorithm would stop here and immediately return the
                  improving move's parameters //;
return $(\Delta Z^*, k^*, n^*)$
```

Algorithm 5.10: *Find-Cross-Move*

```
Input  : Current solution $\boldsymbol{T}$.
Output: Cost-saving, $\Delta Z^*$; and route, subtrip, and arc positions, $i^*$, $j^*$, $k^*$, $l^*$, $m^*$, $n^*$, of the best
        move.
$\Delta Z^* = 0$;
$i^* = 0$; $j^* = 0$; $k^* = 0$;
$l^* = 0$; $m^* = 0$; $n^* = 0$;
for $i = 1$ to $|\boldsymbol{T}| - 1$ do
    for $j = 1$ to $|\boldsymbol{T}_i|$ do
        for $l = i + 1$ to $|\boldsymbol{T}|$ do
            for $m = 1$ to $|\boldsymbol{T}_l|$ do
                $(\Delta Z^*_{\text{trips}}, k', n') =$ Find-Cross-Subtrip-Move$(\boldsymbol{T}_{i,j}, \boldsymbol{T}_{l,m})$ // Algorithm 5.9 //;
                if $\Delta Z^*_{\text{trips}} < \Delta Z^*$ then
                    $\Delta Z^* = \Delta Z^*_{\text{trips}}$;
                    $i^* = i$; $j^* = j$; $k^* = k'$;
                    $l^* = l$; $m^* = m$; $n^* = n'$;
                    // for first-move, the algorithm would stop here and immediately return the
                      improving move's parameters //;
// should $k^* = 0$, then no improving move could be found //;
return $(\Delta Z^*, i^*, j^*, k^*, l^*, m^*, n^*, u^*, v^*)$
```

of route $\boldsymbol{T}_i$ up-to and including arc $T_{i,j,k}$, such that:

$$\mathscr{Z}_{i,j,k} = \begin{cases} \mathscr{Z}_{i,j,k-1} + D(T_{i,j,k-1}, T_{i,j,k}) + w(T_{i,j,k}) & \text{if } 1 < k < |\boldsymbol{T}_{i,j}|, \\ \mathscr{Z}_{i,j,k-1} + D(T_{i,j,k-1}, T_{i,j,k}) + \lambda & \text{if } k = |\boldsymbol{T}_{i,j}|, \\ \mathscr{Z}_{i,j-1,|\boldsymbol{T}_{i,j-1}|} + D(|\boldsymbol{T}_{i,j-1}|, T_{i,j,k}) & \text{if } k = 1 \text{ and } j > 1, \\ 0 & \text{if } k = 1 \text{ and } j = 1. \end{cases} \tag{5.4}$$

If a *cross* move is to be performed on $T_{i,j,k} = u$ and $1 < k < |\boldsymbol{T}_{i,j}|$, then the cost of the partial routes $\boldsymbol{T}'_{i_a}$ and $\boldsymbol{T}'_{i_b,j}$ are given by

$$Z(\boldsymbol{T}'_{i_a}) = \mathscr{Z}_{i,j,k-1}, \tag{5.5}$$

$$Z(\boldsymbol{T}'_{i_b}) = Z(\boldsymbol{T}_i) - \mathscr{Z}_{i,j,k-1} - D(T_{i,j,k-1,i,j,k}). \tag{5.6}$$

In cases where $k = 1$ or $k = |\boldsymbol{T}_{i,j}|$, Equations (5.5) and (5.6) can be used to calculate the cost of the partial route up-to the first arc task preceding u, and from the first arc task following u. Using Equations (5.5) and (5.6) as well as ΔZ_{link_i} and ΔZ_{link_j}, as specified in Figure 5.12, the route duration feasibility of a move can be efficiently determined.

As a last acceleration mechanism, we scanned the subtrip $\boldsymbol{T}_{l,m}$ backwards for *cross* moves. Say that a move is to be performed at $T_{i,j,k}$. If $\mathscr{L}_{i,j,k-1} + load(\boldsymbol{T}_{l,m}) - \mathscr{L}_{l,m,n-1} > Q$, then $\mathscr{L}_{i,j,k-1} + load(\boldsymbol{T}_{l,m}) - \mathscr{L}_{l,m,n-2} > Q$ since $\mathscr{L}_{l,m,n-2} < \mathscr{L}_{l,m,n-1}$. All moves between $T_{i,j,k}$ and $T_{l,m}$ up-to $T_{l,m,n}$ will also be infeasible and need not be considered. As such, when scanning $\boldsymbol{T}_{l,m}$ backwards, the search for moves between $T_{i,j,k}$ and $\boldsymbol{T}_{l,m}$ can terminate when the capacity on the resulting $\boldsymbol{T}'_{i,j}$ subtrip is exceeded. Using the same logic, the route $\boldsymbol{T}_{l,m}$ and its subtrips were also scanned backwards and the route duration constraint monitored.

5.3.5 Two-opt-1

The last move operator that we adapt for the MCARPTIF is *two-opt-1* that focusses on a single route. As illustrated in Figure 5.1, the move inverts a route segment from $T_{i,j,k}$ to $T_{i,l,m}$ within a single route. As further shown in the figure, any *two-opt-1* move can be decomposed into two *cross* moves, allowing for the reuse of the *cross* implementations. This does, however, require that the entire route $\boldsymbol{T}_i$ be inverted prior to the move being evaluated. The cost of the move is then the sum of the two *cross* moves, plus the cost difference between the normal and inverted route segment. The cost difference can be calculated using $\mathscr{Z}_{i,j,k}$ and $\mathscr{Z}^{(inv)}_{i,j',k'}$, with the latter defined for the inverted route $T^{(inv)}_i$. For a move between $T_{i,j,k} = u$ and $T_{i,l,m} = v$, the cost of inverting the route segment from k to m can be calculated as follows:

$$Z_{k\to m} = \mathscr{Z}_{i,l,m} - \mathscr{Z}_{i,j,k-1} - D(T_{i,j,k-1}, T_{i,j,k}), \tag{5.7}$$

$$Z^{(inv)}_{k'\to m'} = \mathscr{Z}^{(inv)}_{i,l',m'} - \mathscr{Z}_{i,j',k'-1} - D(T_{i,j,k-1}, T_{i,j,k}), \tag{5.8}$$

$$\Delta Z_{inv} = Z^{(inv)}_{k'\to m'} - Z_{k\to m}, \tag{5.9}$$

where j', k', l' and m' are defined as shown in Figure 5.13. The load capacity checks of a *cross* have to be performed if the *two-opt-1* move involves arcs in different subtrips.

Algorithm 5.11 returns the best *two-opt-1* move on a single route. Moves are evaluated between arc-pairs in the same route, and for all routes in the solution. If the solution consists of a single route, the move will be evaluated between all arcs in $\boldsymbol{R}_T$. This gives the operator a worst case computational complexity of $\mathcal{O}(|\boldsymbol{R}_T|^2)$ per iteration.

In this section we gave technical descriptions of five move operators for the MCARPTIF. Additional move operators can be adapted for the problem, such as *double-relocate* [6, 51]

Original route $\boldsymbol{T}_i$ with a planned *two-opt* move between arcs $T_{i,1,3} = u_2$ and $T_{i,3,3} = u_7$:

$$\boldsymbol{T}_i \;= \big[[\sigma, u_1, u_2, u_3, \Phi_1], [\Phi_1, u_4, u_5, \Phi_2], [\Phi_2, u_6, u_7, u_8, \Phi_3], [\Phi_3, \sigma]\big]$$

Route $\boldsymbol{T}'_i$ after the *two opt* move has been implemented:

$$\boldsymbol{T}'_i = \big[[\sigma, \underline{u_1}, u'_7, \underline{u'_6}, \Phi_2], [\Phi_2, \underline{u'_5, u'_4}, \Phi_1], [\Phi_1, \underline{u'_3}, u'_2, \underline{u_8}, \Phi_3], [\Phi_3, \sigma]\big] \quad \text{where } u' = \begin{cases} inv(u) & \text{if } inv(u) \neq 0, \\ u & \text{otherwise,} \end{cases}$$

Arcs affected by move

Decomposing a *two-opt-1* move into two *cross* moves

Step 1: Invert the complete route $\boldsymbol{T}_i$ and let $\boldsymbol{T}_i^{(inv)}$ be the result

$$\boldsymbol{T}_i^{(inv)} = \big[[\sigma, u'_8, u'_7, u'_6, \Phi_2], [\Phi_2, u'_5, u'_4, \Phi_1], [\Phi_1, u'_3, u'_2, u'_1, \Phi_3], [\Phi_3, \sigma]\big]$$

Step 2: *Cross* between $\boldsymbol{T}_i$ at position $T_{i,j,k}$, and $\boldsymbol{T}_i^{(inv)}$ at position $T^{(inv)}_{i,m',n'}$ with $m' = |\boldsymbol{T}_i| - m$ and $n' = |\boldsymbol{T}_{i,j}| - n + 1$.

$$\boldsymbol{T}_i = \big[[\sigma, u_1, u_2, u_3, \Phi_1], [\Phi_1, u_4, u_5, \Phi_2], [\Phi_2, u_6, u_7, u_8, \Phi_3], [\Phi_3, \sigma]\big]$$

$$\boldsymbol{T}_i^{(inv)} = \big[[\sigma, u'_8, u'_7, u'_6, \Phi_2], [\Phi_2, u'_5, u'_4, \Phi_1], [\Phi_1, u'_3, u'_2, u'_1, \Phi_3], [\Phi_3, \sigma]\big]$$

Resulting route

$$\boldsymbol{T}'_i = \big[[\sigma, u_1, u'_7, u'_6, \Phi_2], [\Phi_2, u'_5, u'_4, \Phi_1], [\Phi_1, u'_3, u'_2, u'_1, \Phi_3], [\Phi_3, \sigma]\big]$$

Step 3: *Cross* between $\boldsymbol{T}'_i$ at position $T_{i,j',k'}$ with $j' = |\boldsymbol{T}_i| - j$ and $k' = |\boldsymbol{T}_{i,j}| - k + 1$, and $\boldsymbol{T}_i$ at position $T_{i,m,n+1}$.

$$\boldsymbol{T}'_i = \big[[\sigma, u_1, u'_7, u'_6, \Phi_2], [\Phi_2, u'_5, u'_4, \Phi_1], [\Phi_1, u'_3, u'_2, u'_1, \Phi_3], [\Phi_3, \sigma]\big]$$

$$\boldsymbol{T}_i = \big[[\sigma, u_1, u_2, u_3, \Phi_1], [\Phi_1, u_4, u_5, \Phi_2], [\Phi_2, u_6, u_7, u_8, \Phi_3], [\Phi_3, \sigma]\big]$$

Resulting route

$$\boldsymbol{T}''_i \;= \big[[\sigma, u_1, u'_7, u'_6, \Phi_2], [\Phi_2, u'_5, u'_4, \Phi_1], [\Phi_1, u'_3, u'_2, u_8, \Phi_3], [\Phi_3, \sigma]\big]$$

Figure 5.13: Example of the *two-opt-1* move where the route segment between two arcs is inverted. The tasks can be in the same subtrip or different subtrips of the same route. Also shown is its decomposition into two *cross* moves.

Algorithm 5.11: *Find-Two-Opt-1-Move*

```
Input  : Route $\boldsymbol{T}_i$ on which the move is applied, and its inverse $\boldsymbol{T}_i^{(inv)}$.
Output: Cost-saving, $\Delta Z^*$, of the best move; subtrip, and arc positions, $j^*$, $k^*$, $m^*$, and $n^*$, of the
        best move.

$\Delta Z^* = 0$;
$j^* = 0$; $k^* = 0$; $m^* = 0$; $n^* = 0$;
for $j = 1$ to $|\boldsymbol{T}_i| - 1$ do
    for $k = 1$ to $|\boldsymbol{T}_{i,j}|$ do
        for $m = j$ to $|\boldsymbol{T}_i| - 1$ do
            if $m = j$ then $k' = k$;
            else $k' = 1$;
            for $n = k'$ to $|\boldsymbol{T}_{i,m}|$ do
                Using $T_{i,j,k}$, $T_{i,m,n}$, $\boldsymbol{T}_i$ and $\boldsymbol{T}^{(inv)}$, evaluate the two appropriate cross moves as
                  shown in Figure 5.13. Let $\Delta Z_1$ be the cost of the first cross move, and $\Delta Z_2$ of
                  the second;
                Calculate $\Delta Z_{inv}$ as shown in Equations (5.7) to (5.9);
                $\Delta Z = \Delta Z_1 + \Delta Z_2 + \Delta Z_{inv}$;
                if all cross conditions shown in Figures 5.12 and 5.19 to 5.22 are met then
                    if $\Delta Z < \Delta Z^*$ then
                        $\Delta Z^* = \Delta Z$;
                        $j^* = j$; $k^* = k$; $m^* = m$; $n^* = n$;
                        // for first-move, the algorithm would stop here and immediately return
                          the improving move's parameters //;
return ($\Delta Z^*$, $j^*$, $k^*$, $m^*$, $n^*$)
```

as well as more advanced *two-opt-1* and *cross* moves [10, 78]. However, care must be taken when extending the neighbourhood as it increases the computational time of LS. In this thesis we chose to only focus on the five move operators, presented in this section, and critically evaluated their improvement contributions on MCARPTIF instances. We further tested the impact of using a reduced neighbourhood consisting only of three move operators, namely *relocate*, *exchange* and *cross*. Results for the tests are discussed in the next section.

5.4 Computational results

In this section we present computational results for the *first-move* and *best-move* LS setups. Tests were peformed on the *Cen-IF*, *Act-IF* and *Lpr-IF* waste collection benchmark sets which cover a range of realistic instances. The instances are discussed in detail in Chapter 2, Section 2.4.2. The LS setups were used to improve three different starting solutions per instance, generated using the deterministic versions of *Path-Scanning*, *Improved-Merge* and *Efficient-Route-Cluster* under the primary objective to minimise the fleet size, K, and secondly to minimise the solution cost, Z. Details for the heuristics can be found in the previous chapter, in Section 4.3. As part of the constructive heuristic implementations, *Reduce-Vehicles* was applied directly on the constructed solution, before being passed to LS.

The efficiencies of the LS setups were critically evaluated on small and large waste collection instances by measuring the CPU time required by the heuristics to reach local optima. The time was then compared against short, medium and long execution-time-limits of 3, 30 and 60 minutes, respectively. We also evaluated the improvement capabilities of LS by calculating the fractional cost improvement made by LS to the initial solution. The measurement is given by ΔZ^f_{LS}, calculated as

$$\Delta Z^f_{\mathrm{LS}} = \frac{Z\big(\boldsymbol{T}^{(0)}\big) - Z\big(\boldsymbol{T}^{(t)}\big)}{Z\big(\boldsymbol{T}^{(0)}\big)}, \tag{5.10}$$

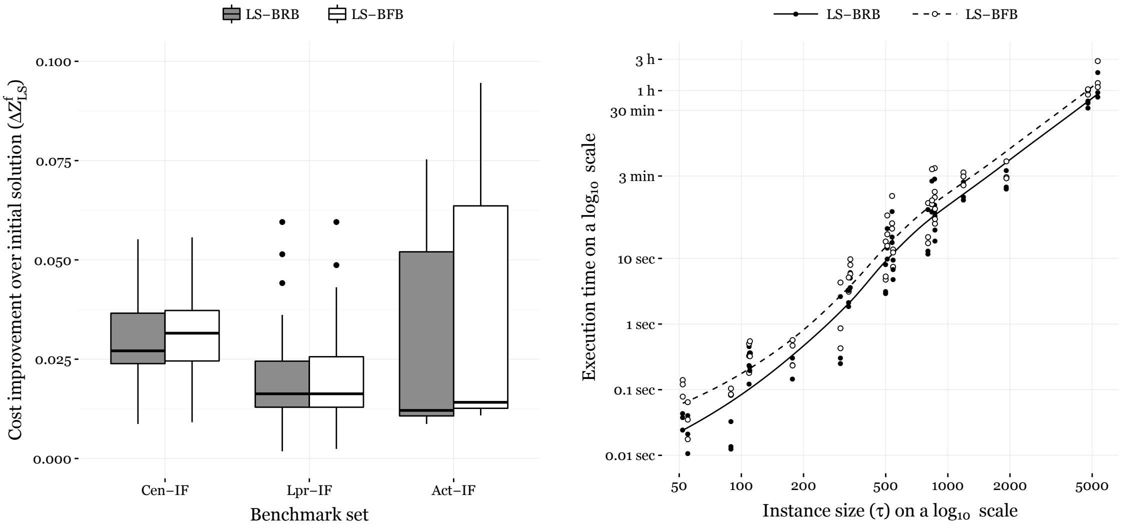
where $Z(\boldsymbol{T}^{(0)})$ is the cost of the initial solution and $Z(\boldsymbol{T}^{(t)})$ is the cost of the local optimum solution returned by LS. Out of the sixty-three initial solutions used for our tests, all of which already had *Reduce-Vehicles* applied to them, fourteen are known to have an excessive fleet-size. To further evaluate the improvement capabilities of LS we counted the number of instances, out of the fourteen, on which the LS setups were able to reduce the fleet size to its best known value.

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 each having 6 cores, and 12 threads and with a 2GHz base frequency. Experiments were run without using programmatic multi-threading or multiple processors.

5.4.1 Best-move local search

Our first tests focussed on two LS setups using the *best-move* strategy. The first setup, referred to as LS-Basic-Full-Best (LS-BFB), used the full move neighbourhood, consisting of *flip*, *relocate*, *exchange*, *cross* and *two-opt-1*, as well as using *Reduce-Vehicles* and *Efficient-IF-Split*, where the latter improves IF visits within each route. The second setup, referred to as LS-Basic-Reduced-Best (LS-BRB), used a reduced neighbourhood, consisting only of *relocate*, *exchange*, *cross* and *Reduce-Vehicles*.

Results for LS-BFB and LS-BFR over the three initial solution per problem instance are shown in Figure 5.14. The fractional cost improvement, ΔZ^f, of the LS setups over the initial solutions are shown in Figure 5.14a. Both LS setups were able to improve



(a) Fraction by which the two basic LS setups improved the costs of initial solutions.

(b) CPU times sand trend-lines of the two basic LS setups versus problem instance size $\tau = |\boldsymbol{R}|$.

Figure 5.14: Results for LS-Basic-Full-Best (LS-BFB) and LS-Basic-Reduced-Best (LS-BFR) setups on waste collection benchmark sets.

on the initial solutions, with the greatest average improvement observed over the *Cen-IF* instances. The improvements over the *Lpr-IF* and *Act-IF* initial solutions was less, although improvements in excess of 5% were observed on a few initial solutions. LS with a full neighbourhood performed marginally better than the reduced neighbourhood version, with the difference in performance being more prominent on the *Cen-IF* instances. In terms of minimising the fleet size, both LS setups were able to reduce the fleet size for eight out of the fourteen excessive-fleet initial solutions. The fleet-size reduction is mainly the result of *Reduced-Vehicles* being applied in each LS iteration, thus allowing it more opportunities to successfully reduce the number of required vehicles.

The computational times of LS-BFB and LS-BFR are shown in Figure 5.14b. Both versions exhibiting quadratic growth as a function of problem size. As expected, LS-BFR was quicker than LS-BFB due to its reduced move neighbourhood. On small and medium instances with up-to 500 required arcs and edges, both versions were relatively quick, capable of reaching local optima in less than 60 seconds. For instances with up-to 2000 arcs and edges, the setups took less than 5 minutes, which are just outside the short time-limits but still well within the medium time-limits. On the large *Cen-IF-b* and *Cen-IF-c* instances the setups struggled, taking in excess of 30 minutes to reach local optima, and in certain cases in excess of three hours. To be useful for metaheuristic applications, LS has to be run either numerous times, or it is required to continue beyond local optima. More efficient LS implementations are thus required.

The most widely used LS acceleration mechanism is to use a *first-move* instead of a *best-move* strategy. With this strategy the move neighbourhood is scanned in a predetermined order and the first improving move found is immediately implemented. To establish a move order we applied the following ranking analysis. For each move type we measured the savings that resulted from its moves over the course of an LS-BFB run. We then calculated the contribution of the moves to ΔZ^f. For example, if LS-BFB 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 as $\frac{200}{10\,000} = 0.02$. The contribution of *exchange* is then calculated as 0.08. To compliment the analysis we also measured the average time required to scan each move-operator's neighbourhood in an LS-BFB iteration.

The contributions of the move-operators per LS-BFB run are shown in Figure 5.15. On all instances *relocate* made the biggest contribution. The contribution of the other operators depended on the instance set, as well as the specific LS setup. *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. This limits the *cross* neighbourhood that only evaluates moves between two different routes. *Two-opt-1* only works on a single route and as a result produces a larger move-neighbourhood on these instances. On all the instances, the other move operators made marginal contributions with *Efficient-IF-Split* being the most effective. *Flip* made very little impact, which may be attributed to *relocate* and *exchange* automatically inverting arcs if the inversion produces a better move.

The average times, per iteration, for the different operators to scan their respective neighbourhoods and return the best move are shown in Figure 5.16. The three main move operators, *relocate*, *exchange* and *cross* exhibit quadratic growth. This is due to their moves being applied between all arcs in $\boldsymbol{T}^{(t)}$. *Exchange* had the longest execution time per iteration, yet its savings contribution was low in comparison to *relocate*. Although not

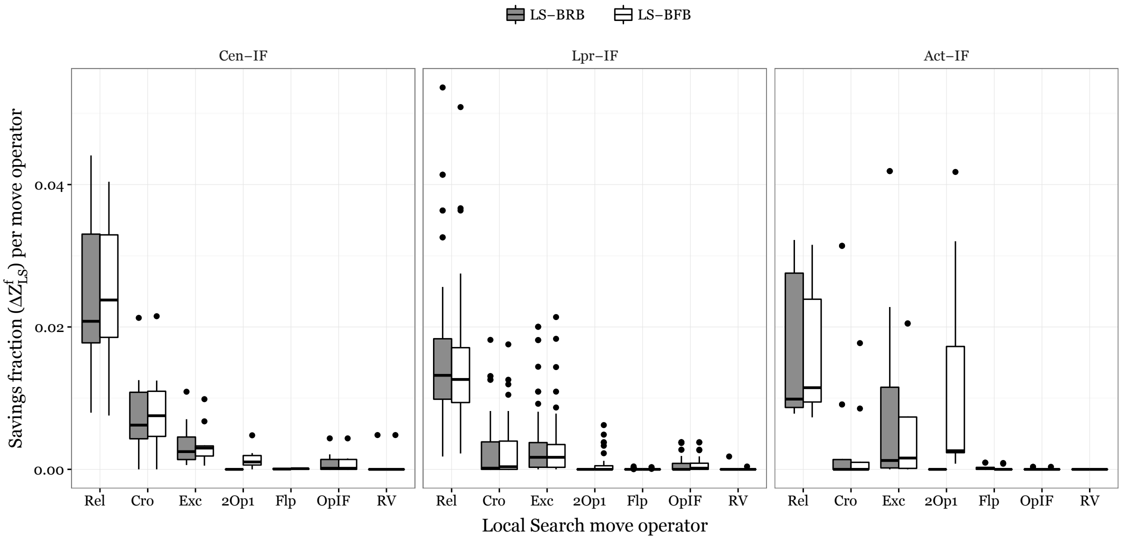


Figure 5.15: Cost saving contributions on waste collection benchmark sets of *Relocate* (Rel), *Cross* (cro), *Exhange* (Exc), *Two-Opt-1* (2Opt1), *Efficient-IF-Split* (OpIF), *Reduce-Vehicles* (RV) and *Flip* (Flp) move operators within the LS-Basic-Full-Best (LS-BFB) and LS-Basic-Reduced-Best (LS-BFR) setups.

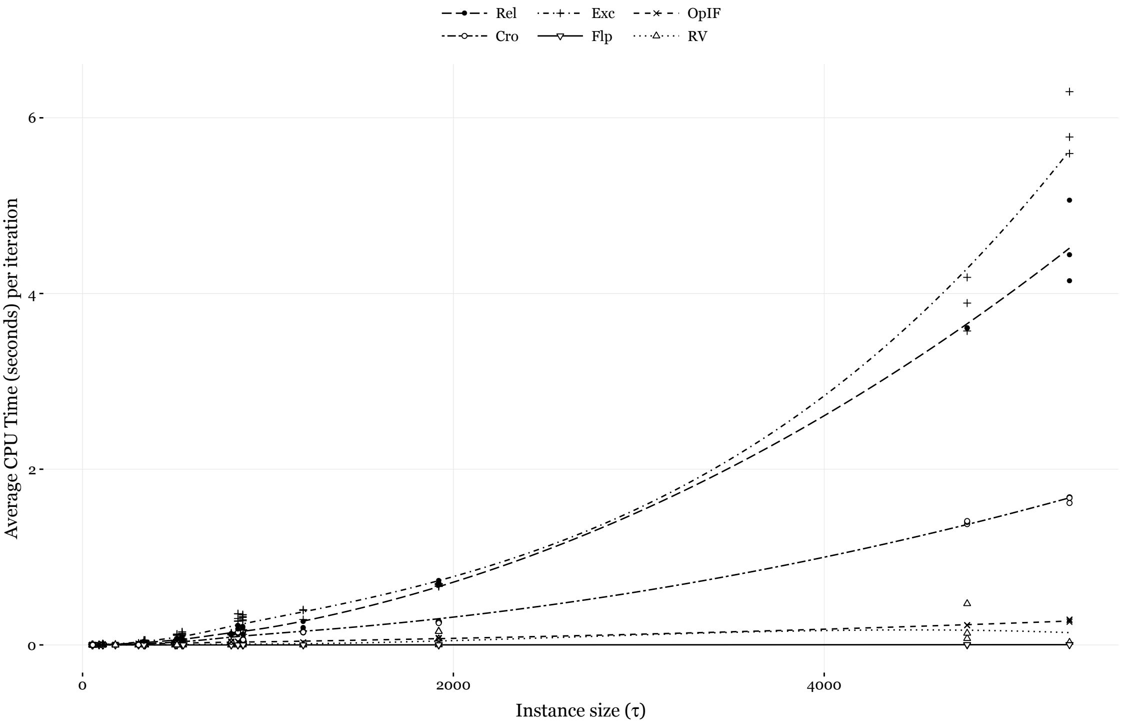


Figure 5.16: Average time required per iteration by each move operator to find and return its best improving move within LS-Basic-Full-Best (LS-BFB) and LS-Basic-Reduced-Best (LS-BFR) setups on waste collection benchmark sets.

considered in this thesis, 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*, *Efficient-IF-Split* and *Reduce-Vehicles* are also very efficient, but as shown in Figure 5.15, they have the smallest contributions to total LS savings.

The aim of the move-operator analysis was to rank the operators for their application within *first-move*. Since the focus of this thesis 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 Figure 5.15. The highest ranking operator is thus *relocate* and the lowest is *flip*. The second aim of the analysis was to identify operators for elimination. Here we relied on the same ranking and identified *two-opt-1*, *Efficient-IF-Split*, and *flip* as elimination candidates. The primary aim of *Reduce-Vehicles* is to reduce the fleet size, not to reduce solution cost, therefore it was not considered for elimination, despite its small contribution to savings.

5.4.2 First-move local search

To evaluate the acceleration potential of *first-move* tests were performed on two LS setups, namely LS-Basic-Full-First (LS-BFF) and LS-Basic-Reduced-First (LS-BRF). The sequence in which the move operators was applied by LS-BFF was *relocate*, *cross*, *exchange*, *two-opt-1* and *flip*, followed by *Efficient-IF-Split* and *Reduce-Vehicles*. LS-BRF used a reduced move-set in the sequence of *relocate*, *cross*, *exchange* and *Reduce-Vehicles*. The two setups were then compared against LS-BFB and LS-BRB.

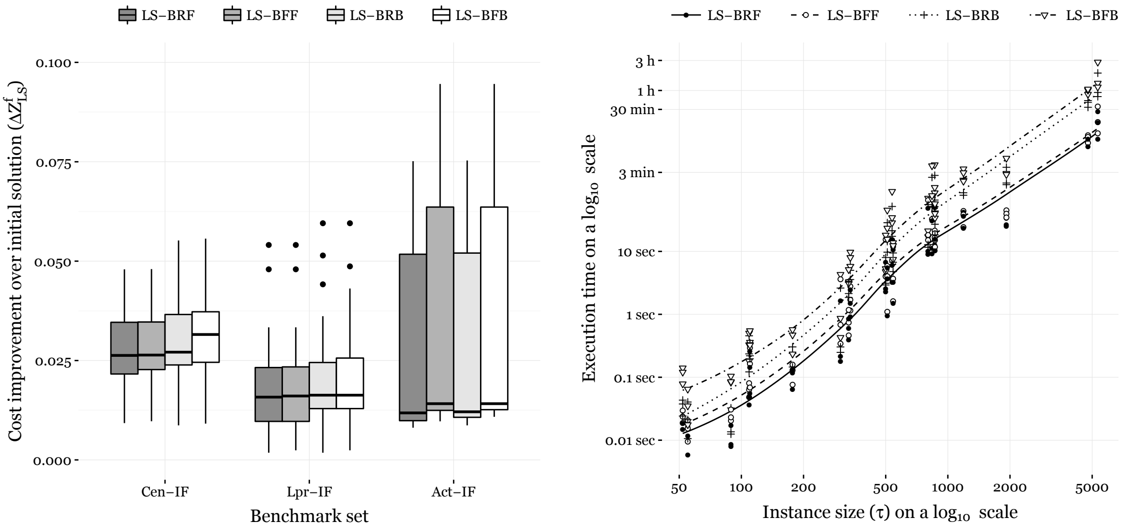
Results for the fleet size reduction capabilities of all the LS setups are shown in Table 5.1. All the setups produced identical results, reducing the fleet sizes for eight out of the fourteen excessive-fleet initial solutions. On the *Cen-IF-b* starting solution produced by *Improved-Merge* the setups were able to reduce the fleet size by eight-vehicles, which represents a 28% reduction. Six local optimum solutions remained with an excess fleet size, with most coming from the *Efficient-Route-Cluster* starting solutions. On these instances different implementations, such as multi-start versions, are required to allow *Reduced-Vehicles* more opportunities to reduce the fleet size.

The cost savings and computational times of the all the setups are shown in Figure 5.17. As shown in Figure 5.17a, the cost savings obtained through LS-BFF and LS-BRF were less than those of LS-BFB and LS-BRB, with LS-BRF performing the worst. On *Act-IF*, LS-BFF performed better than LS-BRB since it evaluates *two-opt-1* moves, which, as discussed earlier, is a major contributor to savings on the benchmark instances. The computational times of each setup to reach local optima are shown in Figure 5.17b. On large problem instances LS-BFF and LS-BRF are significantly faster than LS-BFB and LS-BFB, with LS-BRF being the most efficient. On the largest *Cen-IF* instance, LS-BFF and LS-BRF took at most thirty-minutes to reach local optima, whereas LS-BRB and LS-BFB took in excess of three hours. The LS-BFF and LS-BRF setups can thus be used as-is under medium time-limits, but their increased speed comes at a trade-off in solution quality. On large instances the setups can only be executed a few times, which limits their use within metaheuristic applications. The test results are consistent with the findings of Belenguer et al. [6] on the MCARP and supports their recommendation that more advanced accelerated LS setups be developed for large MCARPs.

Table 5.1: Fleet size reduction capabilities of four local search setups on fourteen excessive-fleet initial solutions.

				K_{BF}			
Instance	K_{BF}	Initial solution	$\lvert\boldsymbol{T}^{(0)}\rvert$	LS-BFB	LS-BRB	LS-BFF	LS-BRF
Cen-IF-b	21	IM	29	8	8	8	8
		PS	22	1	1	1	1
		ERC	22	-	-	-	-
Cen-IF-c	19	PS	20	1	1	1	1
		ERC	20	-	-	-	-
Lpr-IF-a-02	1	ERC	2	-	-	-	-
Lpr-IF-a-04	5	IM	6	1	1	1	1
Lpr-IF-a-05	8	IM	9	1	1	1	1
Lpr-IF-b-02	1	IM	2	-	-	-	-
		PS	2	-	-	-	-
		ERC	2	-	-	-	-
Lpr-IF-b-05	8	IM	9	1	1	1	1
Lpr-IF-c-03	4	IM	5	1	1	1	1
		ERC	5	1	1	1	1
Number of solutions with reduced fleets				*8*	*8*	*8*	*8*

K_{BF}: Best known fleet size; $\lvert\boldsymbol{T}^{0}\rvert$: fleet size of the initial solution; $\lvert\boldsymbol{T}^{(t)}\rvert - K_{BF}$ difference between the local optimum and best known fleet size; IM: *Improved-Merge*; PS: *Path-Scanning*; *ERC*: *Efficient-Route-Cluster*



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

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

Figure 5.17: Comparison of LS-Basic-Full-Best (LS-BFB), LS-Basic-Reduced-Best (LS-BFR), LS-Basic-Full-First (LS-BFF) and LS-Basic-Reduced-First (LS-BRF) setups on waste collection benchmark sets.

5.5 Conclusion

LS is an important and widely applied improvement method for CARPs, and is used as the basic optimisation component of more intelligent metaheuristic improvement methods for the problems. In this chapter four basic LS setups were developed and tested on waste collection benchmark instances. Tests showed that LS is capable of improving MCARPTIF initial solutions by reducing the cost and number of required vehicles of the initial solution. On realistically sized instances, the most efficient LS version with a *first-move* strategy and a reduced move neighbourhood took between five and thirty-minutes to improve an initial solution to its local optimum. To be more effective in situations where LS has to be called numerous times, or when short time-limits are imposed, the LS implementations have to be accelerated, a task made difficult due to the MCARPTIF being asymmetrical and due to additional move-cost calculations resulting from IF visits. Parallelisation may offer opportunities by allowing for the heuristics to be run on multiple instances simultaneously, but acceleration mechanisms would still be needed to speed-up LS per run.

In the next chapter we adapted and implemented advanced LS acceleration mechanisms, originally developed for the Vehicle Routing Problem (VRP) and CARP, to the MCARPTIF. The mechanisms allow LS to be used under medium as well as short execution time-limits on large instances, and are thus ideal candidates for incorporating into advanced metaheuristic methods.

Chapter appendix

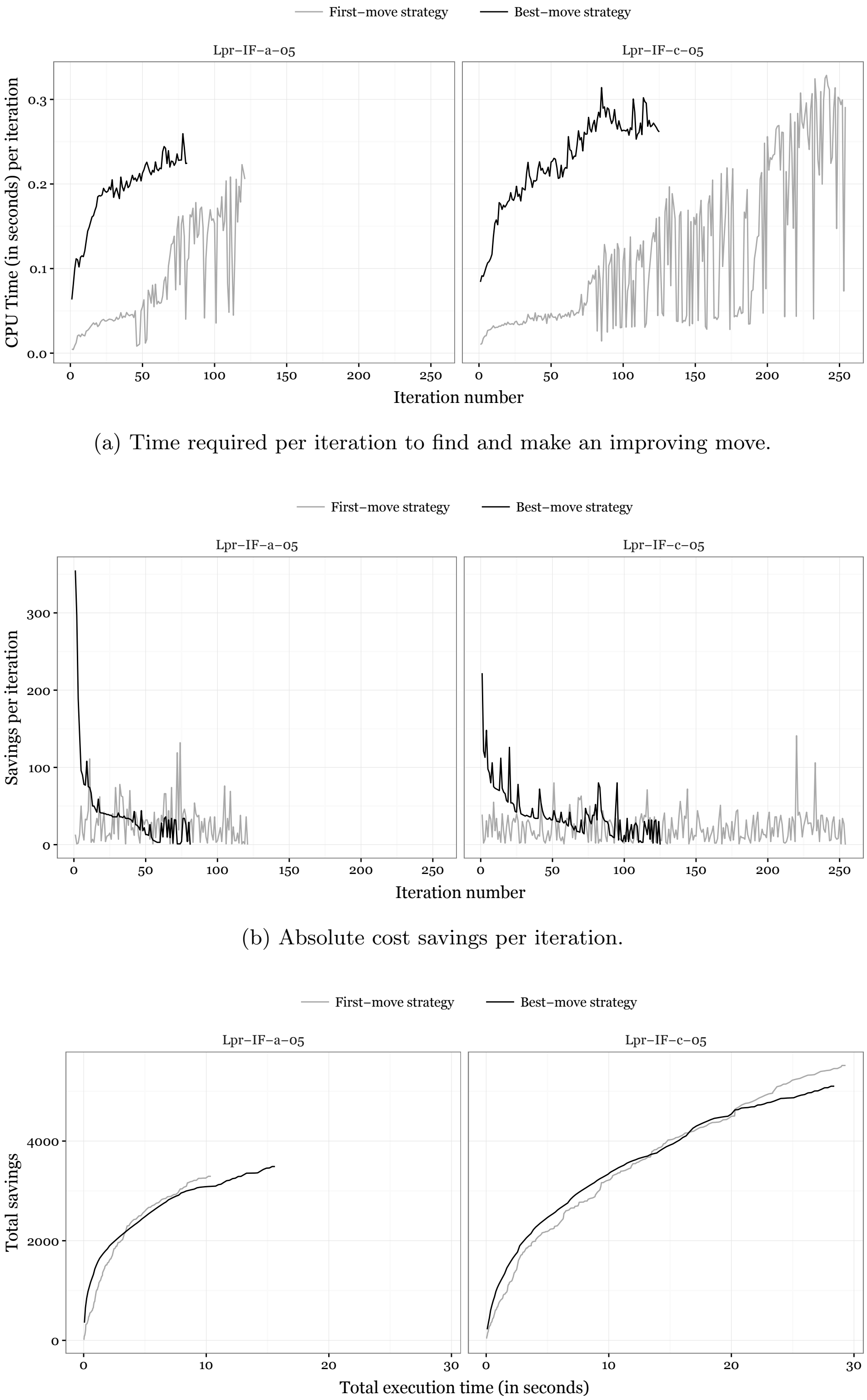
5.A First-move versus best-move strategies

Figure 5.18 shows the execution of two basic *first-move* and *best-move* LS setups, using only the *relocate* move operator, on two problem instances. The time required by the setups to find and return an improving move in each iteration is shown in Figure 5.18a. The execution time of *best-move* is relatively stable since it scans the entire neighbourhood for the best move in each iteration. The variance in its iteration time is due to the pre-move evaluation mechanisms that we implemented in the basic LS setups. The execution time of *first-move* is much more erratic. In the beginning of its execution, *first-move* quickly finds an improving move by virtue of there being more improving moves available. As the search progresses, improving moves become more scarce, and *first-move* has to scan more of the neighbourhood to find them. By the end of the search, the execution of *first-move* is close to that of *best-move* since the entire neighbourhood has to be searched to determine if a local optimum has been reached.

The savings obtained per iteration during the execution of the setups are shown in Figure 5.18b. For each iteration *best-move* finds the best improving move, and as a result, its first moves have the biggest savings. As these moves become exhausted, *best-move* gradually starts to make smaller and smaller improving moves, until no more improving moves are left. *First-move* indiscriminately makes an improving move the moment that it is found. As a result, both small and big improving moves are made throughout its execution, until none are left. The total savings obtained through the setups over their execution-times are shown in Figure 5.18c. On the *Lpr-IF-a-05* instance *best-move* took longer than *first-move* to reach a local optimum. The quality of its local optimum solution was also better. The opposite was observed on *Lpr-IF-c-05* where *first-move* took longer and reached a better local optimum. In general *best-move* is expected to be slower but reach better local optima than *first-move*, but as shown, this is not always the case.

5.B Additional cross moves for the MCARPTIF

The cost calculations and feasibility checks are not given in Figure 5.22, but can be inferred using Figures 5.12 and 5.19 to 5.21.



(a) Time required per iteration to find and make an improving move.

(b) Absolute cost savings per iteration.

(c) Total savings over the execution-time of LS.

Figure 5.18: Illustration of the difference between *best-move* and *first-move* setups using only the *relocate* move operator on two *Path-Scanning* initial solutions.

$\boldsymbol{T}_1 = [[\sigma, u_1, u_2, u_3, \Phi_1], [\Phi_1, u_4, u_5, \Phi_2], [\Phi_2, \sigma]]$ $\boldsymbol{T}_2 = [[\sigma, v_1, v_2, v_3, \Phi_1], [\Phi_1, v_4, v_5, \Phi_2], [\Phi_2, \sigma]]$

If $2 < k < |\boldsymbol{T}_{i,j}|$ and $n = |\boldsymbol{T}_{l,m}|$ then:

Routes after *cross* move: $\boldsymbol{T}_1^{(1)} = [[\sigma, u_1, u_2, \underline{\Phi_1}], [\Phi_1, v_4, v_5, \Phi_2], [\Phi_2, \sigma]]$ $\boldsymbol{T}_2^{(1)} = [[\sigma, v_1, v_2, v_3, \underline{u_3}, \Phi_1], [\Phi_1, u_4, u_5, \Phi_2], [\Phi_2, \sigma]]$

Move costs

$\Delta Z_{split_i} = D(\underset{u_2}{T_{i,j,k-1}}, \underset{u_3}{T_{i,j,k}})$ $\Delta Z_{split_l} = \mu^*(\underset{v_3}{T_{l,m,n-1}}, \underset{v_4}{T_{l,m+1,2}})$

$\Delta Z_{link_i} = \mu^*(\underset{u_2}{T_{i,j,k-1}}, \underset{v_4}{T_{l,m+1,2}})$ $\Delta Z_{link_l} = D(\underset{v_3}{T_{l,m,n-1}}, \underset{u_3}{T_{i,j,k}})$

$\Delta Z = Z_{link_i} - Z_{split_i} + Z_{link_l} - Z_{split_l}$

Partial-routes

$\boldsymbol{T}'_{i_a} = \big[\ldots, [\ldots, T_{i,j,k-1}]\big]$ $\boldsymbol{T}'_{i_b} = \big[[T_{i,j,k}, \ldots], \ldots\big]$ $\boldsymbol{T}'_{l_a} = \big[\ldots, [\ldots, T_{l,m,n-1}]\big]$ $\boldsymbol{T}'_{l_b} = \big[[T_{l,m+1,2}, \ldots], \ldots\big]$

$\boldsymbol{T}'_{1_a} = [[\sigma, u_1, u_2]]$ $\boldsymbol{T}'_{1_b} = [[\underline{u_3}, \Phi_1], [\Phi_1, u_4, u_5, \Phi_2], [\Phi_2, \sigma]]$ $\boldsymbol{T}'_{2_a} = [[\sigma, v_1, v_2, v_3]]$ $\boldsymbol{T}'_{2_b} = [[v_4, v_5, \Phi_2], [\Phi_2, \sigma]]$

Feasibility check:

$Z(\boldsymbol{T}'_{i_a}) + \Delta Z_{link_i} + Z(\boldsymbol{T}'_{l_b}) \leq L$ and

$load(\boldsymbol{T}_{l_a,m}) + load(\boldsymbol{T}_{i_b,1}) \leq Q$ and
$Z(\boldsymbol{T}'_{l_a}) + \Delta Z_{link_l} + Z(\boldsymbol{T}'_{i_b}) \leq L$

Figure 5.19: The cost and feasibility checks when two routes are *crossed* at a position preceded by an arc task, and on the last IF visit in a subtrip.

$\boldsymbol{T}_1 = [[\sigma, u_1, u_2, u_3, \Phi_1], [\Phi_1, u_4, u_5, \Phi_2], [\Phi_2, \sigma]]$ $\boldsymbol{T}_2 = [[\sigma, v_1, v_2, v_3, \Phi_1], [\Phi_1, v_4, v_5, \Phi_2], [\Phi_2, \sigma]]$

If $2 < k < \boldsymbol{T}_{i,j}$ and $n = 1$ then:

Routes after *cross* move: $\boldsymbol{T}_1^{(1)} = [[\sigma, u_1, u_2, \Phi_1], [\underline{\Phi_1}, v_4, v_5, \Phi_2], [\Phi_2, \sigma]]$
$\boldsymbol{T}_2^{(1)} = [[\sigma, v_1, v_2, v_3, \Phi_1], [\Phi_1, \underline{u_3}, \Phi_1], [\Phi_1, u_4, u_5, \Phi_2], [\Phi_2, \sigma]]$

Move costs

$\Delta Z_{split_i} = D(\underset{u_2}{T_{i,j,k-1}}, \underset{u_3}{T_{i,j,k}})$ $\Delta Z_{split_l} = \mu^*(\underset{v_3}{T_{l,m-1,|\boldsymbol{T}_{l,m-1}|-1}}, \underset{v_4}{T_{l,m,2}})$

$\Delta Z_{link_i} = \mu^*(\underset{u_2}{T_{i,j,k-1}}, \underset{v_4}{T_{l,m,2}})$ $\Delta Z_{link_l} = \mu^*(\underset{v_3}{T_{l,m-1,|\boldsymbol{T}_{l,m-1}|-1}}, \underset{u_3}{T_{i,j,k}})$

$\Delta Z = Z_{link_i} - Z_{split_i} + Z_{link_l} - Z_{split_l}$

Partial-routes

$\boldsymbol{T}'_{i_a} = \big[\ldots, [\ldots, T_{i,j,k-1}]\big]$ $\boldsymbol{T}'_{i_b} = \big[[T_{i,j,k}, \ldots], \ldots\big]$ $\boldsymbol{T}'_{1_a} = [[\sigma, u_1, u_2]]$ $\boldsymbol{T}'_{1_b} = [[\underline{u_3}, \Phi_1], [\Phi_1, u_4, u_5, \Phi_2], [\Phi_2, \sigma]]$

$\boldsymbol{T}'_{l_a} = \big[\ldots, [\ldots, T_{l,m-1,|\boldsymbol{T}_{l,m-1}|-1}]\big]$ $\boldsymbol{T}'_{l_b} = \big[[T_{l,m,2}, \ldots], \ldots\big]$ $\boldsymbol{T}'_{2_a} = [[\sigma, v_1, v_2, v_3]]$ $\boldsymbol{T}'_{2_b} = [[\underline{v_4}, v_5, \Phi_2], [\Phi_2, \sigma]]$

Feasibility check:

$Z(\boldsymbol{T}'_{i_a}) + \Delta Z_{link_i} + Z(\boldsymbol{T}'_{l_b}) \leq L$ and $Z(\boldsymbol{T}'_{l_a}) + \Delta Z_{link_l} + Z(\boldsymbol{T}'_{i_b}) \leq L$

Figure 5.20: The cost and feasibility checks when two routes are *crossed* at a position preceded by an arc task, and on the first IF visit in a subtrip.

$\boldsymbol{T}_1 = [[\sigma, u_1, u_2, u_3, \Phi_1], [\Phi_1, u_4, u_5, \Phi_2], [\Phi_2, \sigma]]$ $\boldsymbol{T}_2 = [[\sigma, v_1, v_2, v_3, \Phi_1], [\Phi_1, v_4, v_5, \Phi_2], [\Phi_2, \sigma]]$

If $2 < k < \boldsymbol{T}_{i,j}$ and $n = 2$ then:

Routes after *cross* move: $\boldsymbol{T}_1^{(1)} = [[\sigma, u_1, u_2, \underline{v_4}, v_5, \Phi_2], [\Phi_2, \sigma]]$ $\boldsymbol{T}_2^{(1)} = [[\sigma, v_1, v_2, v_3, \Phi_1], [\Phi_1, \underline{u_3}, \Phi_1], [\Phi_1, u_4, u_5, \Phi_2], [\Phi_2, \sigma]]$

Move costs

$\Delta Z_{split_i} = D(\underset{u_2}{T_{i,j,k-1}}, \underset{u_3}{T_{i,j,k}})$ $\Delta Z_{split_l} = \mu^*(\underset{v_3}{T_{l,m-1,|\boldsymbol{T}_{l,m-1}|-1}}, \underset{v_4}{T_{l,m,2}})$

$\Delta Z_{link_i} = D(\underset{u_2}{T_{i,j,k-1}}, \underset{v_4}{T_{l,m,2}})$ $\Delta Z_{link_l} = \mu^*(\underset{v_3}{T_{l,m-1,|\boldsymbol{T}_{l,m-1}|-1}}, \underset{u_3}{T_{i,j,k}})$

$\Delta Z = Z_{link_i} - Z_{split_i} + Z_{link_l} - Z_{split_l}$

Partial-routes

$\boldsymbol{T}'_{i_a} = \big[\ldots, [\ldots, T_{i,j,k-1}]\big]$ $\boldsymbol{T}'_{i_b} = \big[[T_{i,j,k}, \ldots], \ldots\big]$ $\boldsymbol{T}'_{1_a} = [[\sigma, u_1, u_2]]$ $\boldsymbol{T}'_{1_b} = [[\underline{u_3}, \Phi_1], [\Phi_1, u_4, u_5, \Phi_2], [\Phi_2, \sigma]]$

$\boldsymbol{T}'_{l_a} = \big[\ldots, [\ldots, T_{l,m-1,|\boldsymbol{T}_{l,m-1}|-1}]\big]$ $\boldsymbol{T}'_{l_b} = \big[[T_{l,m,2}, \ldots], \ldots\big]$ $\boldsymbol{T}'_{2_a} = [[\sigma, v_1, v_2, v_3]]$ $\boldsymbol{T}'_{2_b} = [[\underline{v_4}, v_5, \Phi_2], [\Phi_2, \sigma]]$

Feasibility check:

$load(\boldsymbol{T}_{i_a,j}) + load(\boldsymbol{T}_{l_b,1}) \leq Q$ and

$Z(\boldsymbol{T}'_{i_a}) + \Delta Z_{link_i} + Z(\boldsymbol{T}'_{l_b}) \leq L$ and

$Z(\boldsymbol{T}'_{l_a}) + \Delta Z_{link_l} + Z(\boldsymbol{T}'_{i_b}) \leq L$

Figure 5.21: The cost and feasibility checks when two routes are *crossed* at a position preceded by an arc task, and on the first arc following an IF visit.

If $k = |\boldsymbol{T}_{i,j}|$ and $n = |\boldsymbol{T}_{l,m}|$

$$\boldsymbol{T}_1^{(0)} = [[\sigma, u_1, u_2, u_3, \underline{\Phi_1}], [\Phi_1, u_4, u_5, \Phi_2], [\Phi_2, \sigma]] \quad \boldsymbol{T}_2^{(0)} = [[\sigma, v_1, v_2, v_3, \underline{\Phi_1}], [\Phi_1, v_4, v_5, \Phi_2], [\Phi_2, \sigma]]$$

or $k = |\boldsymbol{T}_{i,j}|$ and $n = 1$

$$\boldsymbol{T}_1^{(0)} = [[\sigma, u_1, u_2, u_3, \underline{\Phi_1}], [\Phi_1, u_4, u_5, \Phi_2], [\Phi_2, \sigma]] \quad \boldsymbol{T}_2^{(0)} = [[\sigma, v_1, v_2, v_3, \Phi_1], [\underline{\Phi_1}, v_4, v_5, \Phi_2], [\Phi_2, \sigma]]$$

or $k = 1$ and $n = 1$

$$\boldsymbol{T}_1^{(0)} = [[\sigma, u_1, u_2, u_3, \Phi_1], [\underline{\Phi_1}, u_4, u_5, \Phi_2], [\Phi_2, \sigma]] \quad \boldsymbol{T}_2^{(0)} = [[\sigma, v_1, v_2, v_3, \Phi_1], [\underline{\Phi_1}, v_4, v_5, \Phi_2], [\Phi_2, \sigma]]$$

or $k = 1$ and $n = 2$

$$\boldsymbol{T}_1^{(0)} = [[\sigma, u_1, u_2, u_3, \Phi_1], [\underline{\Phi_1}, u_4, u_5, \Phi_2], [\Phi_2, \sigma]] \quad \boldsymbol{T}_2^{(0)} = [[\sigma, v_1, v_2, v_3, \Phi_1], [\Phi_1, \underline{v_4}, v_5, \Phi_2], [\Phi_2, \sigma]]$$

or $k = 2$ and $n = 2$

$$\boldsymbol{T}_1^{(0)} = [[\sigma, u_1, u_2, u_3, \Phi_1], [\underline{\Phi_1}, u_4, u_5, \Phi_2], [\Phi_2, \sigma]] \quad \boldsymbol{T}_2^{(0)} = [[\sigma, v_1, v_2, v_3, \Phi_1], [\Phi_1, \underline{v_4}, v_5, \Phi_2], [\Phi_2, \sigma]]$$

then:

$$\boldsymbol{T}_1^{(1)} = [[\sigma, u_1, u_2, u_3, \Phi_1], [\Phi_1, v_4, v_5, \Phi_2], [\Phi_2, \sigma]] \quad \boldsymbol{T}_2^{(1)} = [[\sigma, v_1, v_2, v_3, \Phi_1], [\Phi_1, u_4, u_5, \Phi_2], [\Phi_2, \sigma]]$$

If $k = |\boldsymbol{T}_{i,j}|$ and $n = 2$ then:

$$\boldsymbol{T}_1^{(0)} = [[\sigma, u_1, u_2, u_3, \underline{\Phi_1}], [\Phi_1, u_4, u_5, \Phi_2], [\Phi_2, \sigma]] \quad \boldsymbol{T}_2^{(0)} = [[\sigma, v_1, v_2, v_3, \Phi_1], [\Phi_1, \underline{v_4}, v_5, \Phi_2], [\Phi_2, \sigma]]$$

$$\boldsymbol{T}_1^{(1)} = [[\sigma, u_1, u_2, u_3, \underline{v_4}, v_5, \Phi_2], [\Phi_2, \sigma]] \quad \boldsymbol{T}_2^{(1)} = [[\sigma, v_1, v_2, v_3, \underline{\Phi_1}], [\Phi_1, u_4, u_5, \Phi_2], [\Phi_2, \sigma]]$$

If $j = |\boldsymbol{T}_i| - 1$ and $k = |\boldsymbol{T}_{i,j}|$ and $j = 1$ and $n = 2$ then:

$$\boldsymbol{T}_1^{(0)} = [[\sigma, u_1, u_2, u_3, \Phi_1], [\Phi_1, u_4, u_5, \underline{\Phi_2}], [\Phi_2, \sigma]] \quad \boldsymbol{T}_2^{(0)} = [[\sigma, \underline{v_1}, v_2, v_3, \Phi_1], [\Phi_1, v_4, v_5, \Phi_2], [\Phi_2, \sigma]]$$

$$\boldsymbol{T}_1^{(1)} = [[\sigma, u_1, u_2, u_3, \Phi_1], [\Phi_1, u_4, u_5, \underline{v_1}, v_2, v_3, \Phi_1], [\Phi_1, \underline{v_4}, v_5, \Phi_2], [\Phi_2, \sigma]]$$

If $j = |\boldsymbol{T}_i|$ and $k = 1$ and $j = 1$ and $n = 2$ then:

$$\boldsymbol{T}_1^{(0)} = [[\sigma, u_1, u_2, u_3, \Phi_1], [\Phi_1, u_4, u_5, \Phi_2], [\underline{\Phi_2}, \sigma]] \quad \boldsymbol{T}_2^{(0)} = [[\sigma, \underline{v_1}, v_2, v_3, \Phi_1], [\Phi_1, v_4, v_5, \Phi_2], [\Phi_2, \sigma]]$$

$$\boldsymbol{T}_1^{(1)} = [[\sigma, u_1, u_2, u_3, \Phi_1], [\Phi_1, u_4, u_5, \Phi_2], [\Phi_2, \underline{v_1}, v_2, v_3, \Phi_1], [\Phi_1, \underline{v_4}, v_5, \Phi_2], [\Phi_2, \sigma]]$$

Figure 5.22: Further examples of *cross* moves, depending on the positions of $T_{i,j,k}$ and $T_{l,m,n}$. The cost and feasibility checks for the moves can be calculated similar to the moves shown in Figures 5.12 and 5.19 to 5.21.

Chapter 6

Accelerated local search heuristics

In this chapter we develop three acceleration mechanisms that enable Local Search (LS) to better cope with realistically sized waste collection instances. The acceleration mechanisms significantly improve the efficiency of LS, allowing it to be used to improve constructive heuristic solutions even when short execution times are available. In a recent survey on the heuristics for the Capacitated Arc Routing Problem (CARP), Prins [69] state that the development of metaheuristics capable of efficiently tackling huge instances should be investigated in priority. All the current metaheuristics for CARPs rely on some form of LS, either using it directly as a sub-procedure to improve multiple-solutions, or by extending it to escape local optima. Efficient LS implementations are thus critical to achieve the goal of developing practically relevant metaheuristics for waste collection planning.

6.1 Introduction

In the previous chapter we showed that basic LS implementations cannot deal with realistically sized instances within reasonable computing times, with the most efficient setup taking between five and thirty-minutes to improve a single solution to a local optimum. This necessitated us to develop and apply advanced accelerations mechanisms to speed up LS. Three acceleration mechanisms, originally developed for the Vehicle Routing Problem (VRP) and CARP, were adapted to the Mixed Capacitated Arc Routing Problem under Time Restrictions with Intermediate Facilities (MCARPTIF), and tests on large instances show that the mechanisms enable LS to reach local optima within short and medium execution time-limits. The efficiency of the implementations allows them to be applied within multi-start frameworks, whereby multiple initial solutions are improved and the best local optimum solution returned.

In the next section we review LS acceleration mechanisms that have been developed for the CARP and VRP. In Section 6.3 we show how the acceleration mechanisms can be adapted to the MCARPTIF. Computational results for our accelerated LS implementations are presented in Section 6.4, followed by our main research findings and conclusions in Section 6.5.

6.2 Acceleration mechanisms for the CARP and VRP

The computational time required by an exhaustive or near-exhaustive search, such as the basic LS heuristics developed in the previous chapter, increases quadratically with instance size, making the methods impractical when dealing with large instances. This issue is not unique to the MCARPTIF and has been addressed for the CARP and VRP through the development of advanced acceleration mechanisms. In this section we review three of the mechanisms and discuss their applicability to the MCARPTIF. We also review methods that enhance the improvement capability of LS.

6.2.1 Nearest neighbour lists

The first acceleration mechanism that we review is *Nearest-Neighbour-Lists*, the mechanism proposed by Belenguer et al. [6] to improve their Memetic Algorithm. The mechanism was developed by Beullens et al. [10] to improve the efficiency of their Guided Local Search metaheuristic, which is still one of the fastest heuristics for the CARP [69]. *Nearest-Neighbour-Lists* enable LS to scan a promising subset of the full neighbourhood. For each required arc u, a nearest neighbour list, $\boldsymbol{N}_u \subset \boldsymbol{R}/\{u, inv(u)\}$, is established that contains its $\lceil f \times |\boldsymbol{R}| \rceil$ closest required arcs, 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 reduced neighbourhood will not be considered, resulting in LS terminating before reaching local optima. As $f \to 1$ 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. [6] to use *Nearest-Neighbour-Lists* to accelerate LS for the Mixed Capacitated Arc Routing Problem (MCARP), and its successful application to the CARP by Beullens et al. [10], we chose to adapt the mechanism for the MCARPTIF. A downside of the mechanism is that there is trade-off between its acceleration and solution improvement capabilities, which we critically evaluate through tests on the large waste collection problem instances.

In the rest of this section we review mechanisms that accelerate LS without compromising solution quality.

6.2.2 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 an arc is relocated in the same route, all moves not involving that route will be unaffected, yet in most cases the moves will be rescanned. To address this inefficiency Zachariadis and Kiranoudis [92] develop *Static-Move-Descriptors* 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 arcs 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 [92] test the acceleration mechanism on the VRP and show 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 becomes 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 8 times longer per iteration than the accelerated version.

Beullens et al. [10] 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 *gdb* and *bccm* instances, making it difficult to predict what effect the acceleration mechanism will have on large MCARPTIF instances. As such, we chose to only adapt the *Static-Move-Descriptors* of Zachariadis and Kiranoudis [92] 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 that we developed for the MCARPTIF in the previous chapter. 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.

6.2.3 Compounding independent moves

The last acceleration mechanism that we review is *Compound-Independent-Moves*, as applied to the VRP by Ergun et al. [28]. The mechanism 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, and if two improving moves are independent, both can be made in the same iteration, subject to their feasibility when made together. Ergun et al. [28] use this basic concept 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.

There are different methods to determine which independent moves to compound. A greedy approach, which we refer to as *Greedy-Compound-Independent-Moves*, will start with the best move and then continue to the next best improving move that is independent from all previous moves made. LS will then move to the next iteration when no more independent moves are left, whereupon the full neighbourhood is again scanned.

Since *Compounded-Independent-Moves* are seen as a single super-move, a 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 from 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. [28] 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 developed and tested the *Greedy-Compounded-Independent-Moves* mechanism. The mechanism was linked with *Static-Move-Descriptors* as well as *Nearest-Neighbour-Lists* and critically evaluated on large waste collection instances.

6.2.4 Other improvement mechanisms

In addition to the acceleration mechanisms reviewed thus far, a few procedures have been developed for the CARP to extend the improvement capabilities of LS. One such procedure commonly used in LS-based metaheuristics is to allow infeasible moves. The cost of an infeasible move is penalised by a value linked to constraint violations. By allowing infeasible moves, LS may move to unexplored regions of the solution space that cannot be reached via feasible moves alone. It may also allow LS to move through infeasible regions to reach local optima more quickly.

Ghiani et al. [31] develop a Tabu Search metaheuristic for the Capacitated Arc Routing Problem under Time Restrictions with Intermediate Facilities (CARPTIF) that is allowed to make infeasible moves, as well as moves that relocate arcs into new empty routes, thus increasing the fleet size, and moves that include additional Intermediate Facility (IF) visits within a route. The moves also consider relocating an IF visit to a different position in a route, as well as the removal of an IF. Unfortunately the benchmark sets used by the authors are no longer publicly available, making it difficult it compare their approach against our MCARPTIF heuristics.

Brandão and Eglese [12] develop a Tabu Search metaheuristic that also uses infeasible moves. According to Prins [69], their Tabu Search algorithm is still the fastest metaheuristic on standard CARP benchmark sets. The metaheuristic is improved by Mei et al. [59] who develop a global repair operator. The operator scans routes that exceed capacity and moves serviced edges from the routes to other routes in which the same edges are deadheaded. In doing so, the total solution cost remains the same and infeasible routes are made feasible. In this thesis we developed a similar strategy that identifies improving infeasible moves, and through the *Compounded-Independent-Moves* mechanisms indentifies neutral complimentary moves (with zero cost) that will result in the infeasible moves becoming feasible.

In this section we reviewed advanced acceleration mechanisms that have been successfully applied to the CARP and VRP. The acceleration mechanisms that we chose to implement for the MCARPTIF were *Nearest-Neighbour-Lists*, *Static-Move-Descriptors* and *Greedy-Compounded-Independent-Moves*. We also developed procedures that allow LS to make infeasible moves linked with repairing moves.

Our acceleration mechanisms only work with the *relocate*, *exchange* and *cross* move-operators developed in the previous chapter, and which our computational tests showed are the main cost-savings contributors on realistic waste collection instances. Details for the move operators can be found in the previous chapter, in Section 5.3. Although not considered in this thesis, the mechanisms can be easily incorporated into *flip* and *Efficient-IF-Split*. More research is required to incorporate it into *two-opt-1* as it involves route and route segment reversals, which we leave for future work.

6.3 Accelerated and extended local search for the MCARP-TIF

Consistent with our terminology from the previous chapters, 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 list of arc tasks $[T_{i,j,1}, \ldots, T_{i,j,|\boldsymbol{T}_{i,j}|}]$. The pointer function, $inv(u) = v$, is used to return the opposing arc of u. The travel time for the shortest path from arc u to arc v is given by $D(u, v)$ and it is assumed that the shortest path is always followed between consecutive tasks. The cost of route $\boldsymbol{T}_i$ is given as $Z(\boldsymbol{T}_i)$ and the cost of the solution is given as $Z(\boldsymbol{T})$. Let $\boldsymbol{R}$ be the set of all required arc tasks and let $\boldsymbol{R}_T \subseteq \boldsymbol{R}$ be the set of arcs that is currently in solution $\boldsymbol{T}$, excluding dummy arcs. It thus consists of all the arc tasks with $inv(u) = 0$, and one of the opposing arcs, u or u', of a required edge, where $inv(u) = u'$. Lastly, let u and v be two different tasks, which can be in the same or different routes or subtrips.

Unlike the basic LS heuristics that we developed in the previous chapter, the acceleration mechanisms require an arc-focussed search whereby moves are evaluated between sets of arcs. Using *relocate* as an example, moves will be evaluated between $u \in \boldsymbol{R}$ and $v \in \boldsymbol{R}_T$. A specific *relocate* move in which arc $u \in \boldsymbol{R}$ is relocated in-front of arc $v \in \boldsymbol{R}_T$ is referred to as $relocate(u, v)$,[1]. To make our algorithm descriptions more concise, the cost of a move is returned using Algorithm 6.1, and the feasibility of a move is checked using Algorithm 6.2. Both algorithms take as input a move identification parameter, $move_i$, which uniquely identifies the type of move being considered. The location of the arcs have to be determined when calculating the cost of the move, again later when checking if it is feasible, and lastly when implementing the move. This requires a mapping function, $T^{-1}(u) = (i, j, k)$, that maps each arc $u \in \boldsymbol{R}$ 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

Algorithm 6.1: *Move-Cost*

```
Input  : Current solution $\boldsymbol{T}$; move type $move_i$; arcs $u$ and $v$ involved in the move.
Output: Cost of the move $\Delta Z$.
$(i, j, k) = T^{-1}(u)$;
$(l, m, n) = T^{-1}(v)$;
// relocate is assigned a unique identifier, in this case $move_i = 1$ //;
if $move_i = 1$ then
    Using $\boldsymbol{T}_{i,j}$, calculate $\Delta Z_{\text{rem}}$ depending on the relative position of $T_{i,j,k}$ as shown in Chapter 5,
      Figure 5.5;
    Using $u$ and $\boldsymbol{T}_{l,m}$, calculate $\Delta Z_{\text{in}}$ depending on the relative position of $T_{l,m,n}$ as shown in
      Chapter 5, Figure 5.6;
    $\Delta Z = \Delta Z_{\text{rem}} + \Delta Z_{\text{in}}$;
// exchange is assigned a unique identifier, in this case $move_i = 2$ //;
if $move_i = 2$ then
    Using $v$ and $\boldsymbol{T}_{i,j}$, calculate $\Delta Z_{\text{rep}_u}$ depending on the relative position of $T_{i,j,k}$ as shown in
      Chapter 5, Figure 5.9;
    Using $u$ and $\boldsymbol{T}_{l,m}$, calculate $\Delta Z_{\text{rep}_v}$ depending on the relative position of $T_{l,m,n}$ as shown in
      Chapter 5, Figure 5.9;
    $\Delta Z = \Delta Z_{\text{rep}_u} + \Delta Z_{\text{rep}_v}$;
// cross is assigned a unique identifier, in this case $move_i = 3$ //;
if $move_i = 3$ then
    Using $T_{i,j,k}$ and $T_{l,m,n}$, calculate $\Delta Z$ depending on the relative positions of $T_{i,j,k}$ and $T_{l,m,n}$ as
      shown in Chapter 5, Figures 5.12 and 5.19 to 5.22;
return $(\Delta Z)$
```

$$\begin{aligned} T^{-1}(inv(u)) &= T^{-1}(u), && (6.1) \\ &= (i, j, k). && (6.2) \end{aligned}$$

[1] With the arc-focused search, the *relocate* move cannot consider the case of inserting an arc before an IF. To evaluate these moves we developed a *relocate-after* operator, which considers relocating arcs after the last arcs in subtrips. We do not give a detailed description of the operator since it is almost identical to the normal *relocate* operator.

Algorithm 6.2: *Check-Feasibility*

```
Input  : Current solution $\boldsymbol{T}$; move type $move_i$; arcs $u$ and $v$ involved in the move.
Output: Logical flag, $feasible$, for the feasibility of a move.
$(i, j, k) = T^{-1}(u)$;
$(l, m, n) = T^{-1}(v)$;
$feasible = False$;
if $move_i = 1$ then // for a relocate move //
    Using $\boldsymbol{T}$, $(i, j, k)$, $u$ and $(l, m, n)$ check if the relocate conditions shown in Chapter 5, Figure 5.7 are met;
    if all relocate conditions are met then $feasible = True$ ;
if $move_i = 2$ then // for an exchange move //
    Using $\boldsymbol{T}$, $(i, j, k)$, $u$, $(l, m, n)$ and $v$ check if the exchange conditions as shown in Chapter 5, Figure 5.10 are met;
    if all exchange conditions are met then $feasible = True$ ;
if $move_i = 3$ then // for a cross move //
    Using $\boldsymbol{T}$, $(i, j, k)$, $u$, $(l, m, n)$ and $v$ check if the cross conditions shown in Chapter 5, Figures 5.12 and 5.19 to 5.22 are met;
    if all cross conditions are met then $feasible = True$ ;
return ($feasible$)
```

This allows *relocate* and *exchange* to automatically consider arc inversion moves, hence why *relocate* moves are evaluated for $u \in \boldsymbol{R}$, and not just for $u \in \boldsymbol{R}_T$. It is assumed that $T^{-1}(u)$ is automatically updated for all $u \in \boldsymbol{R}$ whenever the solution changes.

As further modifications to our basic LS heuristics, all improving moves, and in some instances of non-improving moves are stored and returned when searching move neighbourhoods, instead of returning a single move for implementation. And lastly, feasibility checks are ignored when searching for improving moves. All the improving moves are returned, and the feasibility checks are performed on a subset of improving candidate moves prior to their implementation.

6.3.1 Nearest neighbour lists

The first acceleration method that we adapted to the MCARPTIF was *Nearest-Neighbour-Lists*, originally developed for the CARP by Beullens et al. [10]. The lists are used in each LS iteration to scan a reduced but promising subset of the full solution neighbourhood. To illustrate why this may be beneficial, consider the example network shown in Figure 6.1. 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 arcs u and v would evaluate the move of relocating arc u from its current service position to be serviced directly before v. Traveling from v to u and then back to v will significantly increase the amount of deadheading in the route of v. From a route design perspective, it would be ideal for arc u to be serviced directly after one of its nearest neighbour arcs. Similarly, if an arc is to be relocated before v, it would be ideal if the arc 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 $u \in \boldsymbol{R} \cup \{\sigma\}$ is formally defined as $\boldsymbol{N}_u \subset \boldsymbol{R}/\{u, inv(u)\}$ and it contains its $s = \lceil f \times m |\boldsymbol{R} \cup \{\sigma\} \rceil$ closest required arcs, where $0 < f \leq 1$ and is user-

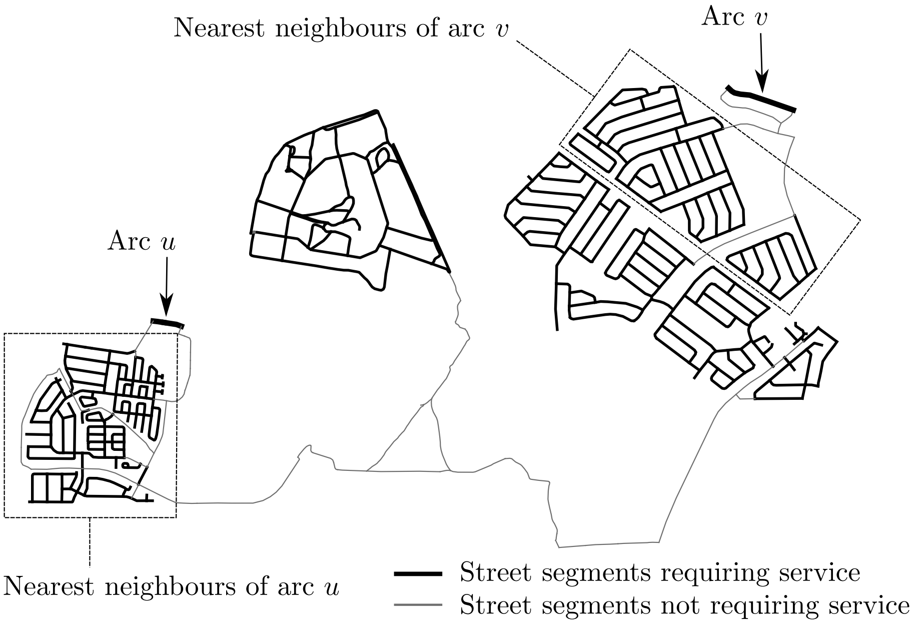

Nearest neighbours of arc v
Arc v
Arc u
Nearest neighbours of arc u
Street segments requiring service
Street segments not requiring service

Figure 6.1: Example of a waste collection area to be serviced. Moves involving u or v should ideally be limited to their respective nearest neighbour arcs.

specified. With our implementation, the lists are sorted in non-decreasing order based on the travel time from u to v, given by $D(u, v)$. For all algorithms 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 arcs 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 6.3. The acceleration, which takes place in lines 4 to 6,

Algorithm 6.3: *Find-Relocate-Moves*

Input : Current solution $\boldsymbol{T}$; savings threshold, $\Delta\overline{Z}$; arcs $\boldsymbol{R}_u$ to be considered for relocation; arcs $\boldsymbol{R}_v$ before which the relocate arcs 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 $\boldsymbol{M}$.

```
1  // both orientations of an edge task can be considered for relocation, so $\boldsymbol{R}_u \subset \boldsymbol{R}$ //;
2  for $u \in \boldsymbol{R}_u$ do
3      $(i, j, k) = T^{-1}(u)$;
4      $s = \lceil f \times |\boldsymbol{R}| \rceil$;
5      $\boldsymbol{R}'_v = \boldsymbol{R}_v \cap \{N_{u,1}, N_{u,2}, \ldots, N_{u,s}\}$ // the possible relocation positions are limited to arcs that
         are in the current solution and that are nearest neighbours of $u$. //;
6      for $v \in \boldsymbol{R}'_v$ do
7          $\Delta Z =$ Move-Cost$(\boldsymbol{T}, move_i = 1, u, v)$ // Algorithm 6.1 //;
8          if $\Delta Z < \Delta\overline{Z}$ then
9              $\boldsymbol{M} = \boldsymbol{M} \cup \{(\Delta Z, move_i = 1, u, v)\}$ ;
10 return $(\boldsymbol{M})$
```

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. Other options are available to limit the move set. For instance, a *relocate* move between $T_{i,j,k} = u$ and $T_{l,m,n} = v$ may be evaluated if the following condition holds:

$$u \in \boldsymbol{N}_{T_{l,m,n-1}}. \tag{6.3}$$

Alternatively, in addition to Equation (6.3), the following two conditions may also be enforced:

$$v \in \boldsymbol{N}_u, \tag{6.4}$$

$$T_{i,j,k+1} \in \boldsymbol{N}_{T_{i,j,k-1}}. \tag{6.5}$$

Since the purpose of the lists is to limit moves to a promising subset of the neighbourhood, and not to limit the moves as much as possible, we only enforced the membership condition of $v \in \{N_{u,1}, \dots, N_{u,s}\}$.

Using Figure 6.1 as an example, assume that v is serviced in a route together with the neighbours of u, and that v is serviced together with the neighbours of u. In this case an *exchange* move between u and v would make sense. However, checking if $u \in \{N_{v,1}, \dots, N_{v,s}\}$ or $v \in \{N_{u,1}, \dots, N_{u,s}\}$ would eliminate the move, unless $f \to 1$. For this reason we instead enforced the membership condition that $v \in \{N_{u_{\text{pre}},1}, \dots, N_{u_{\text{pre}},s}\}$ where $T^{-1}(u) = (i, j, k)$ and $u_{\text{pre}} = T_{i,j,k-1}$ on *exchange* moves. An improving *exchange* move can then be found using Algorithm 6.4.

Algorithm 6.4: *Find-Exchange-Moves*

Input : Current solution $\boldsymbol{T}$; savings threshold, $\Delta\overline{Z}$; arcs $\boldsymbol{R}_u$ and $\boldsymbol{R}_v$ 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 $\boldsymbol{M}$.

```
// both orientations of an edge task can be considered for relocation, so $\boldsymbol{R}_u \subset \boldsymbol{R}$ //;
for $u \in \boldsymbol{R}_u$ do
    $(i, j, k) = T^{-1}(u)$;
    $u_{\text{pre}} = T_{i,j,k-1}$;
    $s = \lceil f \times |\boldsymbol{R}| \rceil$;
    $\boldsymbol{R}'_v = \boldsymbol{R}_v \cap \{N_{u_{\text{pre}},1}, \dots, N_{u_{\text{pre}},s}\}$;
    for $v \in \boldsymbol{R}'_v$ do
        // an exchange between $v$ and $u$ is the same as an exchange between $u$ and $v$, so only one has to be evaluated //;
        $\Delta Z =$ Move-Cost$(\boldsymbol{T}, move_i = 2, u, v)$ // Algorithm 6.1 //;
        if $u < v$ then
            if $\Delta Z < \Delta\overline{Z}$ then
                $\boldsymbol{M} = \boldsymbol{M} \cup \{(\Delta Z, move_i = 2, u, v)\}$ ;
return $(\boldsymbol{M})$
```

The last move operator that we modified was *cross*. A *cross* move between $T_{i,j,k} = u$ and $T_{l,m,n} = v$ will relocate arc v directly after $T_{i,j,k-1}$. The *Nearest-Neighbour-List* membership condition for *exchange* can thus also be used for *cross*. An improving *cross* move can be found using Algorithm 6.5.

6.3.2 Greedily compounding independent moves

The second acceleration method that we adapted for the MCARPTIF was *Independent-Compound-Moves*. In our basic *best-move* LS implementations, all improving moves are

Algorithm 6.5: *Find-Cross-Moves*

Input : Current solution $\boldsymbol{T}$; savings threshold, $\Delta\overline{Z}$; arcs $\boldsymbol{R}_u$ and $\boldsymbol{R}_v$ 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 $\boldsymbol{M}$.

```
// routes can only be crossed at arcs that are currently in the solution, so $\boldsymbol{R}_u \subset \boldsymbol{R}_T$ //;
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
        if $u < v$ then
            $\Delta Z =$ Move-Cost$(\boldsymbol{T}, move_i = 3, u, v)$ // Algorithm 6.1 //;
            if $\Delta Z < \Delta\overline{Z}$ then
                $\boldsymbol{M} = \boldsymbol{M} \cup \{(\Delta Z, move_i = 3, u, v)\}$ ;
return $(\boldsymbol{M})$
```

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 arcs u_1 and v_1 is considered independent from a 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 6.2 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 which involves relocating arc u_2 before arc u_8. The cost of the move is calculated as

$$\Delta Z = D(u_1, u_3) - D(u_1, u_2) - D(u_2, u_3) + D(u_7, u_2) + D(u_2, u_8) - D(u_7, u_8), \quad (6.6)$$

Arcs used in the cost calculation, in addition to u_2 and u_8, are underlined in Figure 6.2. 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 arcs 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 in the same LS iteration. If Move 2 is implemented instead of Move 3, the cost of Move 4 will be unchanged, so Move 2 and 4 could be implemented in the same LS iteration, instead of Move 1 and 3. However, the combined cost of the moves is only -45. Similarly the combined cost of Moves 3 and 4 is only -25. Move 1 and 3 should be compounded into a super-move as it gives the best combined savings.

To identify a sequence of independent moves, we refer to two consecutive arcs, 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 6.2 between arcs u_2 and u_8, the links used to calculate ΔZ are those between the underlined and highlighted arcs, specifically (u_1, u_2), (u_2, u_3) and (u_7, u_8). These links are referred to

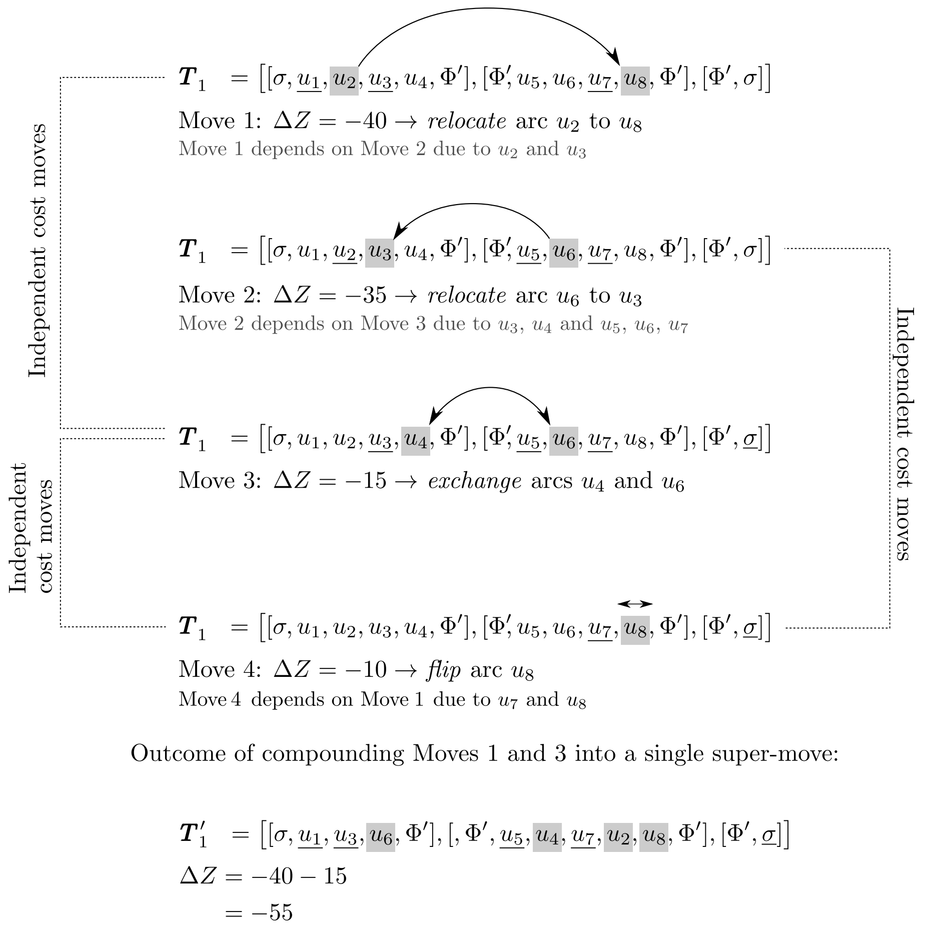

Independent cost moves
$\boldsymbol{T}_1 = [[\sigma, \underline{u_1}, u_2, \underline{u_3}, u_4, \Phi'], [\Phi', u_5, u_6, \underline{u_7}, u_8, \Phi'], [\Phi', \sigma]]$
Move 1: $\Delta Z = -40 \rightarrow$ *relocate* arc u_2 to u_8
Move 1 depends on Move 2 due to u_2 and u_3
$\boldsymbol{T}_1 = [[\sigma, u_1, \underline{u_2}, u_3, u_4, \Phi'], [\Phi', \underline{u_5}, u_6, \underline{u_7}, u_8, \Phi'], [\Phi', \sigma]]$
Move 2: $\Delta Z = -35 \rightarrow$ *relocate* arc u_6 to u_3
Move 2 depends on Move 3 due to u_3, u_4 and u_5, u_6, u_7
Independent cost moves
$\boldsymbol{T}_1 = [[\sigma, u_1, u_2, \underline{u_3}, u_4, \Phi'], [\Phi', \underline{u_5}, u_6, \underline{u_7}, u_8, \Phi'], [\Phi', \underline{\sigma}]]$
Move 3: $\Delta Z = -15 \rightarrow$ *exchange* arcs u_4 and u_6
Independent cost moves
$\boldsymbol{T}_1 = [[\sigma, u_1, u_2, u_3, u_4, \Phi'], [\Phi', u_5, u_6, \underline{u_7}, \overset{\leftrightarrow}{u_8}, \Phi'], [\Phi', \underline{\sigma}]]$
Move 4: $\Delta Z = -10 \rightarrow$ *flip* arc u_8
Move 4 depends on Move 1 due to u_7 and u_8
Outcome of compounding Moves 1 and 3 into a single super-move:
$\boldsymbol{T}_1' = [[\sigma, \underline{u_1}, \underline{u_3}, u_6, \Phi'], [, \Phi', \underline{u_5}, u_4, \underline{u_7}, u_2, u_8, \Phi'], [\Phi', \underline{\sigma}]]$
$\Delta Z = -40 - 15$
$= -55$

Figure 6.2: Four possible improving moves on the example route $\boldsymbol{T}_1$. The arcs between which the moves are applied are highlighted in grey, and arcs used to calculate the costs of the moves are underlined.

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 arcs in the links are removed or replaced, or if an arc is inserted in between the linked arcs.

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

$$(i, j, k) = T^{-1}(u), \tag{6.7}$$

$$pre(u) = \begin{cases} T_{i,j,k-1} \text{ if } k > 2 \text{ or } j = 1, \\ T_{i,j-1,|\boldsymbol{T}_{i,j-1}|-1} \text{ otherwise,} \end{cases} \tag{6.8}$$

$$post(u) = \begin{cases} T_{i,j,k+1} \text{ if } k < |\boldsymbol{T}_{i,j} - 1|, \\ T_{i,j+1,2} \text{ otherwise.} \end{cases} \tag{6.9}$$

Equations (6.7) to (6.9) allow for cost-links to be established between two arcs that are separated by an IF visit. This is necessary when a move involves arcs 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 6.1, and the conditions under which each type of move will break a cost-link can found in Table 6.2. Once $\boldsymbol{M}$ has been populated with information of improving moves, Tables 6.1 and 6.2 can be used to determine which of the moves are independent. The next step is then to compound the moves into a single feasible super-move.

Table 6.1: Cost-links of moves between arcs u and v.

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

$pre(u)$ and $post(u)$ are defined in Equations (6.7) to (6.9).

Table 6.2: Conditions for a move between arcs 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 $v = \theta_{k+1}$

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 [28]. To solve the compounding problem we developed a greedy heuristic that identifies and immediately makes feasible independent moves. A high-level description of the heuristic can be found in Algorithm 6.6. The algorithm takes as input $\boldsymbol{M}$. Each entry in the list, $\pi \in \boldsymbol{M}$, contains critical move information, including the moves cost, ΔZ, a unique identifier for the move type, and the arcs between which the move is applied. The list is sorted 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

Algorithm 6.6: *Greedy-Compound-Independent-Moves*

```
Input  : Current solution $\boldsymbol{T}$; savings threshold, $\Delta\overline{Z}$; savings-list $\boldsymbol{M}$.
Output: Neighbouring solution, $\boldsymbol{T}'$, with independent moves implemented on $\boldsymbol{T}$; total savings,
        $\Delta Z_{\text{total}}$, resulting from the compounded moves.
Let $\boldsymbol{C} = \varnothing$ and $\Delta Z_{\text{total}} = 0$;
Order $\boldsymbol{M}$ from the best to worst improving move;
Let $\boldsymbol{T}' = \boldsymbol{T}$;
for $\pi \in \boldsymbol{M}$ do
    $(\Delta Z, move_i, u, v) = \pi$;
    if $\Delta Z < \Delta\overline{Z}$ then
        if according to Table 6.2 the move does not break any of the cost-links in $\boldsymbol{C}$ then
            $feasible$ =Check-Feasibility($\boldsymbol{T}, u, v$) // Algorithm 6.2 //;
            if $feasible = True$ then
                Add all the move's cost-links shown in Table 6.1 to $\boldsymbol{C}$ ;
                Implement the move on $\boldsymbol{T}'$;
                Let $\Delta Z_{\text{total}} = \Delta Z_{\text{total}} + \Delta Z$;
                // for a pure find-best implementation the heuristic would stop here and
                  immediately return $\boldsymbol{T}$ and $\Delta Z_{\text{total}}$ //;
return ($\boldsymbol{T}$, $\Delta Z_{\text{total}}$)
```

from *all* previous moves that have been implemented in the current LS iteration. 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.

Infeasible moves are included in $\boldsymbol{M}$ since route costs and subtrip loads change as the independent moves are implemented. Moves that are infeasible at the start of the iteration may become feasible later. It is also possible that infeasible moves high-up in $\boldsymbol{M}$ will later-on become feasible. One option is to rescan $\boldsymbol{M}$ from the first entry each time an independent move is made. Alternatively, the moves can be carried over to the next LS iteration 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. The savings list can also be implemented as a priority-queue whereby it is already sorted when passed to Algorithm 6.6. We leave these implementations and their evaluation for future work.

By treating the compounded moves as a single super-move, the general structure of LS using *Greedy-Independent-Compound-Moves* remain the same. The only requirements are that all improving moves be returned through Algorithms 6.3 to 6.5, which allows for *Nearest-Neighbour-Lists* acceleration, and that moves be implemented using Algorithm 6.6.

6.3.3 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 [92] 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, $move_i$; arc u involved in the move; and arc v involved in the move. Information used for the feasibility checks can also be added, such as pre- and post-arcs of u and v, the change in load to the arcs' subroutes, and the change in cost to the arcs' routes. The move information contained in π meets all the requirements for 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 6.3. The list is then sorted and its first, thus best, feasible improving move will be implemented using a modified version of Algorithm 6.6. 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 6.1. After a move is implemented, all other moves whose cost-links have been broken by the move, as defined in Table 6.2, 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 $(v, post(v))$. 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 6.2, a cost-link of the move between u and v would have been broken if $u^* \in \{pre(u), u, post(u), v, post(v)\}$ or if $v^* \in \{u, post(u), post(v)\}$. To update the descriptors, all moves which involve relocating arc $pre(u^*)$, u^*, $post(u^*)$, v^* or $post(v^*)$ have to be rescanned, so too all moves in which an arc is inserted before u^*, $post(u^*)$ or $post(v^*)$. To update the move descriptors, $\boldsymbol{M}$ is scanned and any descriptor with u or $inv(u) \in \{pre(u^*), u^*, post(u^*), v^*, post(v^*)\}$ or $v \in \{u^*, post(u^*), post(v^*)\}$ is removed. Thereafter the descriptors are updated and inserted back into $\boldsymbol{M}$ as follows:

$$\boldsymbol{R}_u = \{pre(u^*), u^*, post(u^*), v^*, post(v^*)\} \tag{6.10}$$

$$\boldsymbol{R}_u^{(inv)} = \{inv(u') : u' \in \boldsymbol{R}_u \text{ and } inv(u') \neq 0\} \tag{6.11}$$

$$\boldsymbol{R}_v = \{u^*, post(u^*), post(v^*)\} \tag{6.12}$$

$$\boldsymbol{M}' = \textit{Find-Relocate-Moves}(\Delta\overline{Z}, \boldsymbol{R}_u \cup \boldsymbol{R}_u^{(inv)}, \boldsymbol{R}_T, \boldsymbol{M}) \tag{6.13}$$

$$\boldsymbol{M}'' = \textit{Find-Relocate-Moves}(\Delta\overline{Z}, \boldsymbol{R}/\boldsymbol{R}_u \cup \boldsymbol{R}_u^{(inv)}, \boldsymbol{R}_v, \boldsymbol{M}') \tag{6.14}$$

$$\boldsymbol{M} = \boldsymbol{M}'' \tag{6.15}$$

in Equations (6.13) and (6.14), Algorithm 6.3 is called to find improving *relocate* moves, but with the move sets $\boldsymbol{R}_u$ and $\boldsymbol{R}_v$ significantly reduced. *Nearest-Neighbour-Lists* may 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 which takes $\mathcal{O}(|\boldsymbol{R}| \times |\boldsymbol{R}_T|)$. Thereafter, the neighbourhood is scanned in $\mathcal{O}(|\boldsymbol{R}| + |\boldsymbol{R}_T|)$.

Since both *Static-Move-Descriptors* and *Greedy-Independent-Compound-Moves* rely on move independence, we use two sets, $\boldsymbol{U}_a$ and $\boldsymbol{U}_b$, to store the arcs of cost-links of implemented moves. The sets are used to determine if candidate moves are independent from all the moves already implemented in the current LS iteration. The sets are also used to update the savings list. A move which was applied between two arcs u and v can either break one or two cost-links on u and v. The first set, $\boldsymbol{U}_a$, is used to check if a move that breaks two cost-links on u or v is independent from previous moves. Such moves include *exchange* where the two cost-links $(pre(u), u)$ and $(u, post(u))$ associated with u as well as the two cost-links $(pre(v), v)$ and $(v, post(v))$ of v are broken; and the remove portion of *relocate* where the two costs links $(pre(u), u)$ and $(u, post(u))$ are broken. The second set, $\boldsymbol{U}_b$, is similarly used to check if a move that only breaks one cost-link on u or v is independent from previous moves. Such moves include *cross* where cost links $(pre(u), u)$ and $(pre(v), v)$ are broken, and the insert portion of *relocate* where the cost-link $(pre(v), v)$ is broken. When a move is implemented, its cost-link arcs can be added to $\boldsymbol{U}_a$ and $\boldsymbol{U}_b$ using Algorithm 6.7. Using $\boldsymbol{U}_a$ and $\boldsymbol{U}_b$, Algorithm 6.8 can then be used to check if a move is independent from all the already implemented moves.

Algorithm 6.7: *Update-Move-Dependence-Arc-Sets*

```
Input  : Move information, $\pi$; dependent arc set, $\boldsymbol{U}_a$, for two-cost-link arc changes; dependent arc
         set, $\boldsymbol{U}_b$, for one-cost-link arc changes.
Output: Updated sets, $\boldsymbol{U}_a$ and $\boldsymbol{U}_b$, with arcs from cost-links of move $\pi$ added to the sets.
$(\Delta Z, move_i, u, v, ) = \pi$;
if $move_i = 1$ then // if it's a relocate move //
    $\boldsymbol{R}'_{\text{rem}} = \{pre(u), u, post(u), v, post(v)\}$;
    $\boldsymbol{R}_{\text{rem}} = \boldsymbol{R}'_{\text{rem}} \cup \{inv(u') : u' \in \boldsymbol{R}'_{\text{rem}}$ and $inv(u') \neq 0\}$;
    $\boldsymbol{U}_a = \boldsymbol{U}_a \cup \boldsymbol{R}_{\text{rem}}$;
    $\boldsymbol{U}_b = \boldsymbol{U}_b \cup \{u, post(u), post(v)\}$;
if $move_i = 2$ then // if it's an exchange move //
    $\boldsymbol{R}'_{\text{exc}} = \{pre(u), u, post(u), pre(v), v, post(v)\}$;
    $\boldsymbol{R}_{\text{exc}} = \boldsymbol{R}'_{\text{exc}} \cup \{inv(u') : u' \in \boldsymbol{R}'_{\text{exc}}$ and $inv(u') \neq 0\}$;
    $\boldsymbol{U}_a = \boldsymbol{U}_a \cup \boldsymbol{R}_{\text{exc}}$;
    $\boldsymbol{U}_b = \boldsymbol{U}_b \cup \{u, post(u), v, post(v)\}$;
if $move_i = 3$ then // if it's a cross move //
    $\boldsymbol{U}_a = \boldsymbol{U}_a \cup \{pre(u), u, pre(v), v\}$;
    $\boldsymbol{U}_b = \boldsymbol{U}_b \cup \{u, v\}$;
return $(\boldsymbol{U}_a, \boldsymbol{U}_b)$
```

is independent from all the already implemented moves.

Algorithm 6.8: *Check-Move-Independence*

```
Input  : Move information, $\pi$; dependent arc set $\boldsymbol{U}_a$ for two-cost-link changes; dependent arc set
         $\boldsymbol{U}_b$ for one-cost-link changes.
Output: Logical flag, independent, for the independence of a move from previous moves.
$(\Delta Z, move_i, u, v, ) = \pi$;
$independent = True$;
if $move_i = 1$ then // if it's a relocate move //
    if $u \in \boldsymbol{U}_a$ or $inv(u) \in \boldsymbol{U}_a$ or $v \in \boldsymbol{U}_b$ or $inv(v) \in \boldsymbol{U}_b$ then $independent = False$ ;
if $move_i = 2$ then // if it's an exchange move //
    if $u \in \boldsymbol{U}_a$ or $inv(u) \in \boldsymbol{U}_a$ or $v \in \boldsymbol{U}_a$ or $inv(v) \in \boldsymbol{U}_a$ then $independent = False$ ;
if $move_i = 3$ then // if it's a cross move //
    if $u \in \boldsymbol{U}_b$ or $inv(u) \in \boldsymbol{U}_b$ or $v \in \boldsymbol{U}_b$ or $inv(v) \in \boldsymbol{U}_b$ then $independent = False$ ;
return $(independent)$
```

To formally link all three acceleration mechanisms developed in this chapter, moves can be compounded using Algorithm 6.9 and the savings list can be updated using Algorithm 6.10. The structure of the accelerated LS heuristic, with all three mechanisms, is then as shown in Algorithm 6.11.

Algorithm 6.9: *Greedy-Compound-Moves*

```
Input  : Current solution, $\boldsymbol{T}$; savings threshold, $\Delta\overline{Z}$; savings-list, $\boldsymbol{M}$.
Output: Neighbouring solution, $\boldsymbol{T}'$, with independent moves implemented on $\boldsymbol{T}$; total savings,
        $\Delta Z_{\text{total}}$, resulting from the compounded moves; dependent arc set $\boldsymbol{U}_a$ for two-cost-link
        changes; dependent arc set $\boldsymbol{U}_b$ for one-cost-link changes.
$\boldsymbol{U}_a = \varnothing$;
$\boldsymbol{U}_b = \varnothing$;
$\Delta Z_{\text{total}} = 0$;
$\boldsymbol{T}' = \boldsymbol{T}$;
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
        $independent$ =Check-Move-Independence($\pi$,$\boldsymbol{U}_a$,$\boldsymbol{U}_b$) // Algorithm 6.8 //;
        if $independent = True$ then
            $feasible$ =Check-Feasibility($\boldsymbol{T}'$, $u, v$) // Algorithm 6.2 //;
            if $feasible = True$ then
                ($\boldsymbol{U}_a$,$\boldsymbol{U}_b$) =Update-Move-Dependence-Arc-Sets($\pi$,$\boldsymbol{U}_a$,$\boldsymbol{U}_b$) // Algorithm 6.7 //;
                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 $\Delta Z_{\text{total}}$ //;
return ($\boldsymbol{T}$', $\Delta Z_{\text{total}}$, $\boldsymbol{U}_a$, $\boldsymbol{U}_b$)
```

6.3.4 Extending the move neighbourhood

The last improvement that we applied to LS was to extend the move neighbourhoods generated by *cross*, *relocate* and *exchange*. Extending the neighbourhoods increases the computational time of LS, therefore the extensions were exclusively used by our accelerated LS implementations.

The first extension that we implemented was to allow *cross* to swap the end portions of subtrips, instead of the end portions of entire routes. When a *cross* move is applied between $T_{i,j,k} = u$ and $T_{l,m,n} = v$, where

$$\boldsymbol{T}_i = \left[\ldots, \boldsymbol{T}_{i,j-1}, [\ldots, T_{i,j,k-2}, T_{i,j,k-1}, \underline{T_{i,j,k}}, T_{i,j,k+1}, \ldots], \boldsymbol{T}_{i,j+1}, \ldots\right], \tag{6.16}$$

$$\boldsymbol{T}_l = \left[\ldots, \boldsymbol{T}_{l,m-1}, [\ldots, T_{l,m,n-2}, T_{l,m,n-1}, \underline{T_{l,m,n}}, T_{l,m,n+1}, \ldots], \boldsymbol{T}_{l,m+1}, \ldots\right], \tag{6.17}$$

the end result would then be

$$\boldsymbol{T}'_i = \left[\ldots, \boldsymbol{T}_{i,j-1}, [\ldots, T_{i,j,k-2}, T_{i,j,k-1}, \underline{T_{l,m,n}}, T_{l,m,n+1}, \ldots], \boldsymbol{T}_{i,j+1}, \ldots\right], \tag{6.18}$$

$$\boldsymbol{T}'_l = \left[\ldots, \boldsymbol{T}_{l,m-1}, [\ldots, T_{l,m,n-2}, T_{l,m,n-1}, \underline{T_{i,j,k}}, T_{i,j,k+1}, \ldots], \boldsymbol{T}_{l,m+1}, \ldots\right]. \tag{6.19}$$

Note that $\boldsymbol{T}_{i,j+1}$ remains in $\boldsymbol{T}'_i$ and $\boldsymbol{T}_{l,m+1}$ in $\boldsymbol{T}'_l$. The move consists of two compounded *cross* moves between u and v, and $T_{i,j+1,2}$ and $T_{l,m+1,2}$, and requires that all four arcs be added to $\boldsymbol{U}_a$ and $\boldsymbol{U}_b$. The move may be applied between different routes as well as different subtrips in the same route. In our implementations, we only evaluated the subtrip *cross* move between $T_{i,j,k} = u$ and $T_{l,m,n} = v$ if $T_{i,j+1,1} = T_{i,m+1,1}$, in which case the cost of the

Algorithm 6.10: *Update-Savings-List*

```
Input  : Dependent arc set, $\boldsymbol{U}_a$, for two-cost-link arc changes; dependent arc set, $\boldsymbol{U}_b$, for
         one-cost-link arc changes; move savings list, $\boldsymbol{M}$; move-cost threshold, $\Delta\overline{Z}$.
Output: Updated move savings list, $\boldsymbol{M}'$

$\boldsymbol{M}' = \varnothing$;
for $\pi \in \boldsymbol{M}$ do
    $independence = $ Check-Move-Independence$(\pi, \boldsymbol{U}_a, \boldsymbol{U}_b)$ // Algorithm 6.8 //;
    if $independence = True$ then
        $\boldsymbol{M}' = \boldsymbol{M}' \cup \{\pi\}$

$\boldsymbol{M}' = $ Find-Relocate-Moves$(\Delta\overline{Z}, \boldsymbol{U}_a, \boldsymbol{R}_T, \boldsymbol{M}')$ // Algorithm 6.3 //;
$\boldsymbol{M}' = $ Find-Relocate-Moves$(\Delta\overline{Z}, \boldsymbol{R}/\boldsymbol{U}_a, \boldsymbol{U}_b, \boldsymbol{M}')$;

$\boldsymbol{M}' = $ Find-Exchange-Moves$(\Delta\overline{Z}, \boldsymbol{U}_a, \boldsymbol{R})$ // Algorithm 6.4 //;
$\boldsymbol{M}' = $ Find-Exchange-Moves$(\Delta\overline{Z}, \boldsymbol{R}/\boldsymbol{U}_a, \boldsymbol{U}_a, \boldsymbol{M}')$;

$\boldsymbol{M}' = $ Find-Cross-Moves$(\Delta\overline{Z}, \boldsymbol{U}_b, \boldsymbol{R}_T)$ // Algorithm 6.5 //;
$\boldsymbol{M}' = $ Find-Cross-Moves$(\Delta\overline{Z}, \boldsymbol{R}_T/\boldsymbol{U}_b, \boldsymbol{U}_b, \boldsymbol{M}')$;
return $(\boldsymbol{M}')$
```

Algorithm 6.11: *Accelerated-Local-Search*

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

 1 $t = 0$;
 2 $\boldsymbol{M} = \varnothing$;
 3 $\boldsymbol{M} = $ Find-Relocate-Moves$(\boldsymbol{T}^{(0)}, \Delta\overline{Z}, \boldsymbol{R}, \boldsymbol{R}_T, \boldsymbol{M})$ // Algorithm 6.3 //;
 4 $\boldsymbol{M} = $ Find-Exchange-Moves$(\boldsymbol{T}^{(0)}, \Delta\overline{Z}, \boldsymbol{R}, \boldsymbol{R}, \boldsymbol{M})$ // Algorithm 6.4 //;
 5 $\boldsymbol{M} = $ Find-Cross-Moves$(\boldsymbol{T}^{(0)}, \Delta\overline{Z}, \boldsymbol{R}_T, \boldsymbol{R}_T, \boldsymbol{M})$ // Algorithm 6.5 //;
 6 repeat
 7     Use Reduce-Vehicles on $\boldsymbol{T}^{(t)}$ to reduce the fleet, and let $\boldsymbol{T}'$ be the result;
 8     if $|\boldsymbol{T}'| < |\boldsymbol{T}^{(t)}|$ then // the fleet size has been reduced //
 9         Set $\boldsymbol{T}^{(0)} = \boldsymbol{T}'$ and return to line 1;

10     if $\boldsymbol{M} \neq \varnothing$ then
11         $(\boldsymbol{T}', \Delta Z_{\text{total}}, \boldsymbol{U}_a, \boldsymbol{U}_b) = $ Greedy-Compound-Moves$(\boldsymbol{T}^{(t)}, \Delta\overline{Z}, \boldsymbol{M})$ // Algorithm 6.9 //;
12         if $\Delta Z_{total} < \Delta\overline{Z}$ then
13             Set $\boldsymbol{T}^{(t+1)} = \boldsymbol{T}'$;
14             $\boldsymbol{M}'' = $ Update-Savings-List$(\boldsymbol{U}_a, \boldsymbol{U}_b, \boldsymbol{M}, \Delta\overline{Z})$ // Algorithm 6.10 //;
15             $\boldsymbol{M} = \boldsymbol{M}''$;
16             $t = t + 1$;

17         else a feasible move could not be found;

18     else an improving move could not be found;
19 until a feasible improving move could not be found;
20 return $(\boldsymbol{T}^{(t)})$
```

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. Recall that when finding the best *cross* move between arcs u and v, the conditions $i \neq j$ or $j \neq m$ must hold since the move has to be applied between two different routes or subtrips. 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 $[T_{i,j,k_1}, \ldots, T_{i,k_2-1}]$ and $[T_{i,n_1}, \ldots, T_{i,n_2-1}]$. In this case the original route

$$\boldsymbol{T}_{i,j} = [\ldots, T_{i,j,k_1-1}, \underline{T_{i,j,k_1} \ldots, T_{i,k_2-1}}, T_{i,k_2}, \ldots, T_{i,n_1-1}, \underline{T_{i,n_1}, \ldots, T_{i,n_2-1}}, T_{i,n_2}, \ldots], \tag{6.20}$$

will become

$$\boldsymbol{T}'_{i,j} = [\ldots, T_{i,j,k_1-1}, \underline{T_{i,n_1}, \ldots, T_{i,n_2-1}}, T_{i,k_2}, \ldots, T_{i,n_1-1}, \underline{T_{i,j,k_1}, \ldots, T_{i,k_2-1}}, T_{i,n_2}, \ldots]. \tag{6.21}$$

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 load constraints are 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 arc 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* and *infeasible-compound* moves we used a greedy approach, similar to *Greedy-Compound-Moves*, 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 if they are independent from *all* the previous moves made in the LS iteration. Each subtrip group is then scanned through this process.

The same approach is followed for infeasible moves. First, infeasible subtrip moves are grouped according to subtrips together with complimentary non-improving *relocate* moves that removes arcs 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, and which is independent from the infeasible and previous moves, both are implemented. The process then repeats for the second best infeasible move, and is applied to all subtrip groups.

Both *double-cross* and *infeasible-compound* may consider pairing improving and non-improving moves, as long as the combined moves result in an improvement. The move neighbourhood can thus be extended by adding non-improving moves to $\boldsymbol{M}$. Since $\boldsymbol{M}$ is sorted in each LS iteration, its size can be limited by specifying a threshold saving $\Delta\overline{Z}$, which is passed to Algorithms 6.3 to 6.5. Only moves with $\Delta Z < \Delta\overline{Z}$ are then added to $\boldsymbol{M}$. 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.

For our LS implementation we searched for and applied *double-cross* and *infeasible-compound* only once LS reached a local optima. If an improving *double-cross* and *infeasible-compound* was found, the savings list was updated and LS reverted to the normal accelerated search, otherwise it terminated.

6.4 Computational results

The aim of this chapter was to develop and evaluate efficient LS heuristics that can be used to improve the initial solutions within short and medium time-limits, and that can be extended to develop efficient metaheuristics for the MCARPTIF. For the majority of our computational tests we relied on the *Act-IF*, *Lpr-IF* and *Cen-IF* waste collection benchmark sets, as they cover a range of realistic instances. The *Cen-IF* instances are especially important as they are consistent in size with the very large problem instances found in practice. All sets are discussed in detail in Chapter 2, Section 2.4.2.

Computational tests were performed on four accelerated LS setups, and their results compared against the basic setups of the previous chapter. For each accelerated setup, *Static-Moved-Descriptors* and *Nearest-Neighbour-Lists* were always applied, as was *Nearest-Neighbour-Lists* that limit moves to an arc's f nearest neighbours. Two of the setups made the *best-move* at each iteration, whereas the other two employed the *Greedy-Independent-Compound-Moves* mechanism. The setups were divided between those that relied on the *relocate*, *cross*, *exchange* and *Reduced-Vehicles* move operators, and those who also evaluated *double-cross* and *infeasible-compound* moves. A summary of the different LS setups and their acronyms, including those from the previous chapter, are shown in Table 6.3.

All LS setups 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 each having 6 cores, and 12 threads and with a 2GHz base frequency. Experiments were run without using programmatic multi-threading or multiple processors.

For the first part of our tests we analysed the execution time of, and the savings obtained through the accelerated LS heuristics on three deterministic initial solutions per benchmark instance. The solutions were generated using *Path-Scanning*, *Improved-Merge* and *Efficient-Route-Cluster* with their primary objective set to minimise the fleet size. Detailed descriptions of the heuristics can be found in Chapter 4, Section 4.3. The aim of the tests was to determine if the acceleration mechanisms improve the efficiency of LS, and to determine if their implementation reduces the quality of the local optima at which LS terminates. As an output of the first tests we identified dominated LS setups and eliminated then from further testing. For the second part of our tests we linked the remaining LS setups with the randomised multi-start constructive heuristics, thereby allowing LS to improve multiple initial solutions per instance and return the best improved solution. The aim of the tests was to find the best constructive heuristic and LS

Table 6.3: The main local search versions and their respective setups tested for the MCARPTIF.

Version	Move neighbourhood	Move strategy	Acronym
Basic (B) (Chapter 5)	Full (F)	Best-Move (B)	BFB
		First-Move (F)	BFF
	Reduced (R)	Best-Move (B)	BRB
		First-Move (F)	BRF
Accelerated (A) (Chapter 6)	Reduced (R)	Best-Move (B)	ARB-f
		Greedy-Independent-Compound-Moves (G)	ARG-f
	Extended (E)	Best-Move (B)	AEB-f
		Greedy-Independent-Compound-Moves (G)	AEG-f

Accelerated setups are linked with *Static-Move-Descriptors* and *Nearest-Neighbour-Lists* with f nearest neighbours; for setups where f is not specified the level $f = 1$ was used. For the reduced neigbourhoods, *relocate*, *cross*, *exchange* and *Reduce-Vehicles* are applied. For the extended move neighbourhoods, *double-cross* and *infeasible-compound* are also applied.

setup combination, to be used under short and medium execution times, and to identify candidate setups that can extended to develop efficient metaheuristics for the MCARPTIF.

6.4.1 Analysis of acceleration mechanisms and extended move neighbourhoods

For our first tests we analysed the efficiency of the accelerated LS setups by measuring the CPU time required by the setups to reach local optima. To compare the improvement capabilities of different setups we further measured the fractional cost improvement made by LS to an initial solution, calculated as

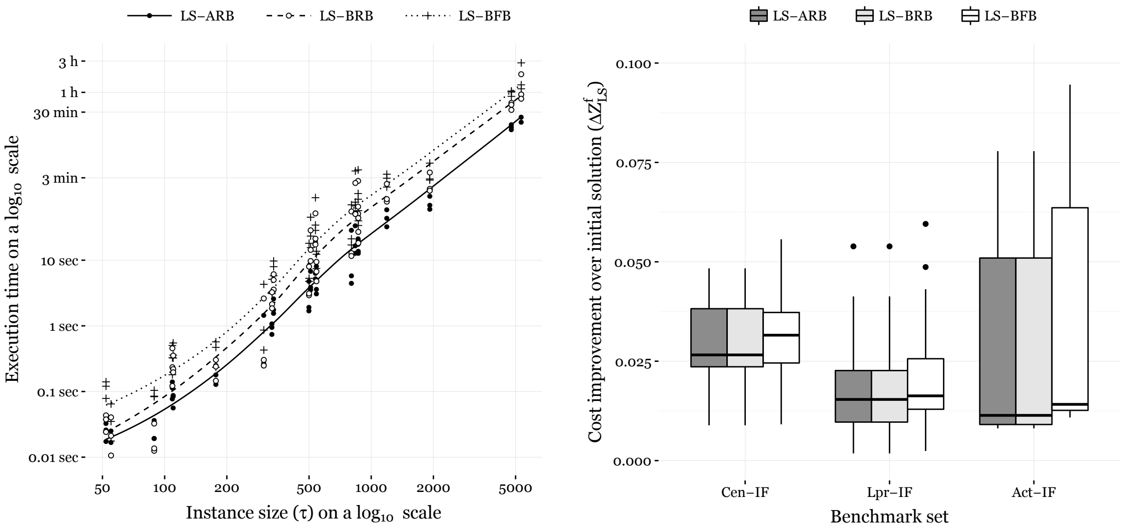
$$\Delta Z^{f}_{\mathrm{LS}} = \frac{Z\big(\boldsymbol{T}^{(0)}\big) - Z\big(\boldsymbol{T}^{(t)}\big)}{Z\big(\boldsymbol{T}^{(0)}\big)}, \tag{6.22}$$

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 LS. This measurement was taken over three initial solutions per problem instance, generated using the *Path-Scanning*, *Improved-Merge* and *Efficient-Route-Cluster* heuristics. Out of the sixty-three initial solutions for the *Act-IF*, *Lpr-IF* and *Cen-IF* instances, fourteen are known to have an excessive fleet-size, this after *Reduce-Vehicles* were applied to the solutions. To analyse the fleet reduction capabilities of LS we counted the number of instances, out of the fourteen, on which the LS setups were able to reduce the fleet size up-to its known minimum size.

The accelerated setups employ three mechanisms, namely, *Static-Move-Descriptors*, *Greedy-Compound-Independent-Moves* and *Nearest-Neighbour-Lists*. Our first tests only focussed on *Static-Move-Descriptors*. Thereafter we tested the setup using *Static-Move-Descriptors* as well as *Greedy-Compound-Independent-Moves*, and lastly we tested the setups using all three mechanisms.

Static-Move-Descriptors

An advantage of *Static-Move-Descriptors* is that it does not effect solution quality. An LS setup will reach the same local optima, with or without its application. As such, the mechanism can always be applied in combination with other acceleration mechanisms without compromising solution quality, assuming that it does in fact accelerate the search. To test this assumption the LS-Accelerated-Reduced-Best (LS-ARB) setup with $f = 1$ was evaluated against LS-Basic-Full-Best (LS-BFB) and LS-Basic-Reduced-Best (LS-BRB) from the previous chapter. Results for the tests are shown in Figure 6.3. The execution times of the setups are shown in Figure 6.3a which confirms that LS-ARB is significantly quicker than both LS-BRB and LS-BFB, particularly on the large *Cen-IF* instances. On the large instances, the execution time of LS-ARB was about half-that of LS-BRB, and as shown in Figure 6.3b the cost savings of the two setups were identical. The vehicle fleet reduction of the setups were also identical, whereby K was reduced to its known minimum on eight of the fourteen excessive-fleet initial solutions. Based on these results we chose to always use *Static-Move-Descriptors* within the accelerated setups. We also eliminated LS-BRB from further testing since LS-ARB is quicker and produces the same local optima.



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

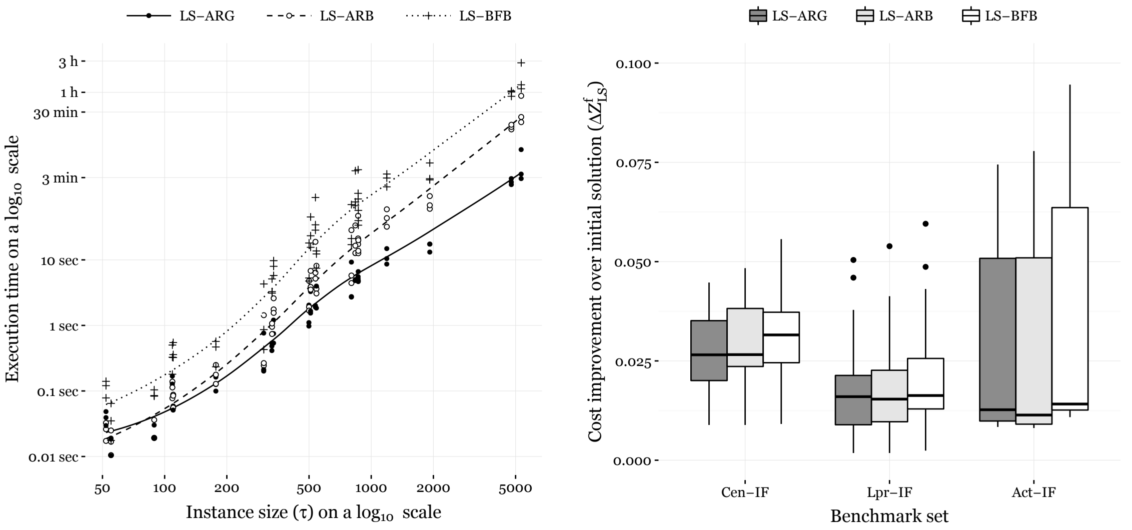
(b) Fraction by which the LS setups improved initial solutions.

Figure 6.3: Comparison of LS-Accelerated-Reduced-Best (LS-ARB), LS-Basic-Reduced-Best (LS-BRB) and LS-Basic-Full-Best (LS-BFB) on waste collection benchmark sets.

The acceleration effect of the *Static-Move-Descriptors* was not as significant as those observed by Zachariadis and Kiranoudis [92] on similarly sized VRP instances, indicating that there is still room for improvement. One such improvement, which Zachariadis and Kiranoudis [92] 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.

Greedy-Compound-Independent-Moves

The second acceleration mechanism that we evaluated was *Greedy-Compound-Independent-Moves*. To test the effect of the mechanism, the LS-Accelerate-Reduced-Greedy (LS-ARG) setup that employs the mechanism in conjunction with *Static-Move-Descriptors* was tested against LS-ARB and LS-BFB on the waste collection instances. Results for the tests are shown in Figure 6.4. As shown in Figure 6.4a, the computational time of LS-ARG is much lower than the times of the other setups, especially on large instances where it took less than three minutes to reach local optima on all but two of the largest *Cen-IF* instances. The difference in computational times between LS-ARG and LS-ARB increases with problem size, indicating that LS-ARG has better scalability. The reason for this is discussed in more detail in Section 6.A at the end of the chapter.



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

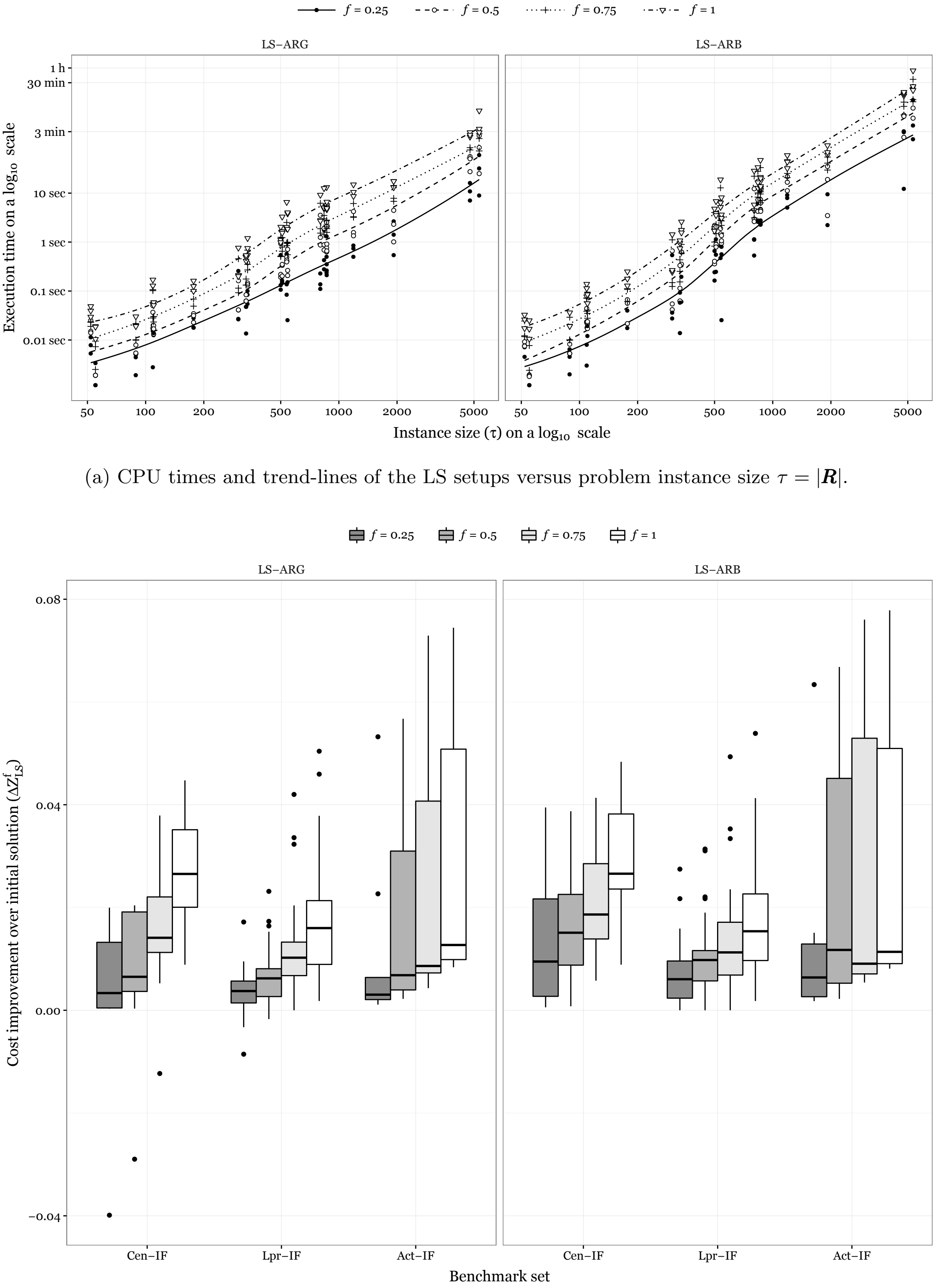
(b) Fraction by which the LS setups improved initial solutions.

Figure 6.4: Comparison of LS-Accelerated-Reduced-Greedy (LS-ARG), LS-Accelerated-Reduced-Best (LS-ARB) and LS-Basic-Full-Best (LS-BFB) on waste collection benchmark sets.

As shown in Figure 6.4b, the improvement of LS-ARG is very similar to that of LS-ARB, outperforming the setup on *Cen-IF* and *Act-IF*. LS-ARB had slightly better improvements on *Cen-IF*, saving it from being completely dominated. The vehicle fleet reduction of LS-ARG was also identical to that of LS-ARB and LS-BFB, in that the fleet size was reduced to its known minimum on eight of the fourteen excessive-fleet initial solutions.

Nearest-Neighbour-Lists

The third acceleration mechanism that we tested was *Nearest-Neighbour-Lists*. The mechanism reduces the move neigbourhood by limiting moves between arcs to a fraction f of the closest neighbours. For the computational tests, LS-ARG and LS-ARB were tested at three levels, namely $f \in \{0.25, 0.5, 0.75\}$, and compared against the previously tested setups with $f = 1$. Results for the setups are shown in Figure 6.5. As shown in Figure 6.5a, reducing the move neighbourhood significantly reduced the computational times of the setups. At $f = 0.25$, LS-ARG took less than 80 seconds on the large *Cen-IF* instances to reach local optima. However, as shown in Figure 6.5b, the reduction in computational times comes at a price, with the coinciding savings of the setups being inversely correlated to f. Some of the LS-ARG-f setups have negative savings, meaning they increased the



(a) CPU times 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 6.5: Comparison of LS-Accelerated-Reduced-Greedy (LS-ARG) and LS-Accelerated-Reduced-Best (LS-ARB) at four *Nearest-Neighbour-Lists* f-levels on waste collection benchmark sets.

cost of the initial solutions. This was as a result of their fleet reductions. During their execution, *Reduce-Vehicles* was able to reduce the fleet size, but by doing so it increased solution cost. The LS-ARG-f setups improved the solution cost thereafter to local optima, but the local optima still had higher costs than the initial solutions. Table 6.4 shows the fleet sizes of the local optima solution on the excessive-fleet initial solutions. Unlike the previous acceleration mechanisms, the nearest-neighbour lists negatively affected the fleet-reduction capabilities of the setups. LS-ARG-0.25 and LS-ARG-0.5 could only reduce the fleet sizes of three solutions, whereas LS-ARB-0.25 and LS-ARB-0.5 could at least reduce four solutions.

Linking *Nearest-Neighbour-Lists* with *Static-Move-Descriptors* and *Greedy-Compound-Independent-Moves* allows LS to be used under short execution time-limits. However, as shown Figure 6.5b and Table 6.4, its application reduces solution quality in correlation to lower f values. The setups were therefore subjected to further testing.

Extended move neighbourhood

The last setups that we tested used an extended neighbourhood when searching for improving moves. In addition to the *relocate*, *exchange*, *cross* and *Reduce-Vehicle* 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 need not be improving, as long as the combined moves result in an improvement on the current solution. Recall that the savings list is sorted and scanned in each iteration to identify complimentary 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 it 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 at the first LS iterations. 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 6.6 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 prevalent. Of interest is the amount of neutral moves, which in most cases outnumbers the improving moves. In an MCARPTIF solution, required arcs and edges are often dead-headed in routes. When a move results in the arc 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 to one, thereby allowing LS to only return improving and neutral moves. An advantage of this approach is that it keeps the savings-list relatively short, but more 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 6.6 there are usually a significant number of neutral moves available.

To evaluate the impact of the extended neighbourhood, LS-Accelerated-Extended-Greedy and LS-Accelerated-Extended-Best were tested on the three starting solutions for the waste collection sets. The results for the setups in comparison to LS-ARG and LS-ARB with a reduced neighbourhood are shown in Figure 6.7. The extended neighbourhood increased the computational time of both setups, more so on the mid-sized *Lpr-IF* instances (Figure 6.7a). On the large *Cen-IF* instances the computational times required to find lo-

Table 6.4: Fleet size reduction, ΔK_{LS}, of LS-ARG-f and LS-ARB-f setups on fourteen excessive-fleet initial solutions.

				LS-ARG				LS-ARB			
Instance	K_{BF}	Initial	$\lvert\boldsymbol{T}^{(0)}\rvert$	$f = 0.25$	$f = 0.5$	$f = 0.75$	$f = 1$	$f = 0.25$	$f = 0.5$	$f = 0.75$	$f = 1$
Cen-IF-b	21	IM	29	7	7	7	8	7	7	7	8
		PS	22	-	-	1	1	-	1	1	1
		ERC	22	-	-	-	-	-	-	-	-
Cen-IF-c	19	PS	20	-	-	1	1	1	1	1	1
		ERC	20	-	-		-	-	-	-	-
Lpr-IF-a-02	1	ERC	2	-	-	-	-	-	-	-	-
Lpr-IF-a-04	5	IM	6	1	1	1	1	1	1	1	1
Lpr-IF-a-05	8	IM	9	1	1	1	1	1	1	1	1
Lpr-IF-b-02	1	IM	2	-	-	-	-	-	-	-	-
		PS	2	-	-	-	-	-	-	-	-
		ERC	2	-	-	-	-	-	-	-	-
Lpr-IF-b-05	8	IM	9	-	-	-	1	-	1	1	1
Lpr-IF-c-03	4	IM	5	-	-	-	1	-	-	-	1
		ERC	5	-	-	-	1	-	-	-	1
Number of solutions with reduced fleets				*3*	*3*	*5*	*8*	*4*	*6*	*6*	*8*

K_{BF}: Best known fleet size; Initial: Constructive heuristic version; $\lvert\boldsymbol{T}^{0}\rvert$: fleet size of the initial solution.

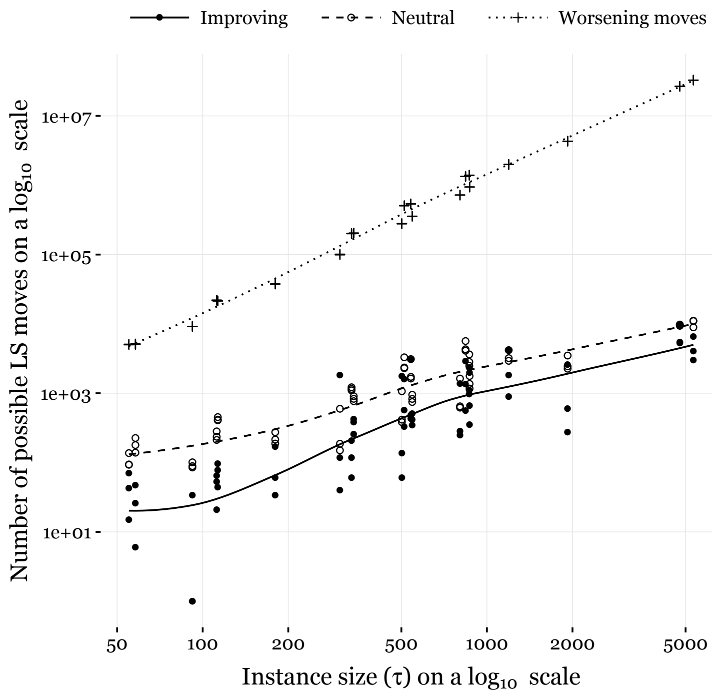


Figure 6.6: Move landscape analysis at the first iteration of LS with a reduced neighbourhood on waste collection benchmark sets.

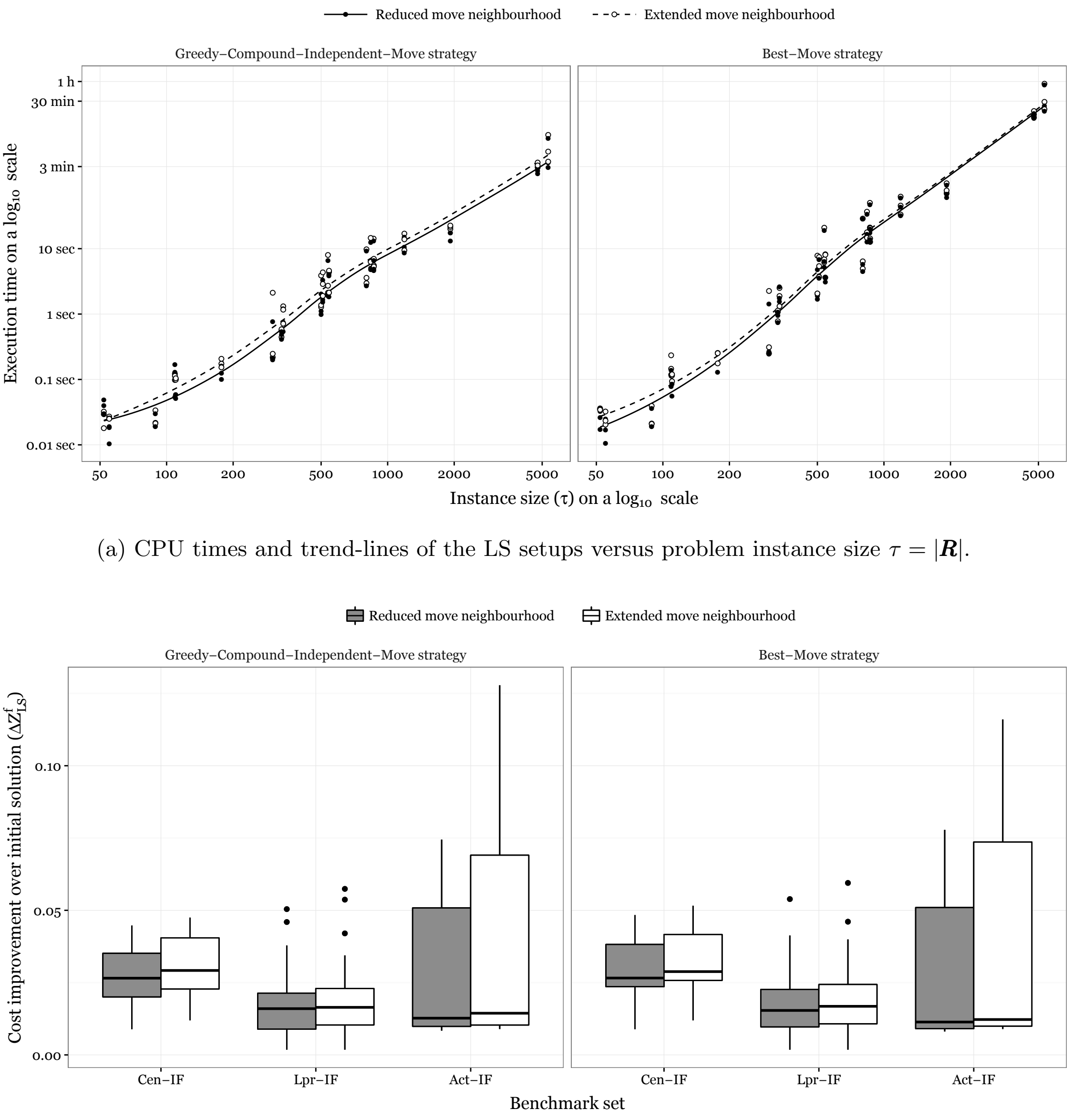
cal optima were relatively close. Better costs savings were obtained through the extended neighbourhood, especially on the *Cen-IF* instances (Figure 6.7b), but it had no impact on vehicle fleet reduction, with LS-AEB and LS-AEG producing identical fleet sizes to those of LS-ARB and LS-ARG. True to its purpose, the extended neighbourhood allowed the LS setups to reach better local optima, and as expected, it increased the computational times of the setups in doing so.

The initial tests on the three 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 neighbourhood. In the rest of this section we critically evaluate and compare all the setups, with the aim to identify and eliminate setups that are both slower and produce worse solutions than other setups.

6.4.2 Domination analysis

Thus far we have evaluated the efficiency and improvement capabilities of thirty-two accelerated LS setups, including those with different nearest-neighbour f levels. In most cases there is a direct trade-off between the solution quality and computational efficiency of the setups. The same holds for the four basic LS setups developed and tested in the previous chapter. Tests on LS-Basic-Full-Best (LS-BFB) showed that it is too slow for medium execution time-limits, with execution times in excess of 30 minutes on large *Cen-IF* instances. As a result, it was eliminated from further testing. LS-BFB was also eliminated since it was dominated by LS-Accelerated-Reduced-Best (LS-ARB); both produce the same local optima, but LS-ARB is much quicker.

The domination of setups on the waste collection benchmark sets, similar to that of LS-ARB over LS-BRB, was then used to identify and eliminate sub-performing setups



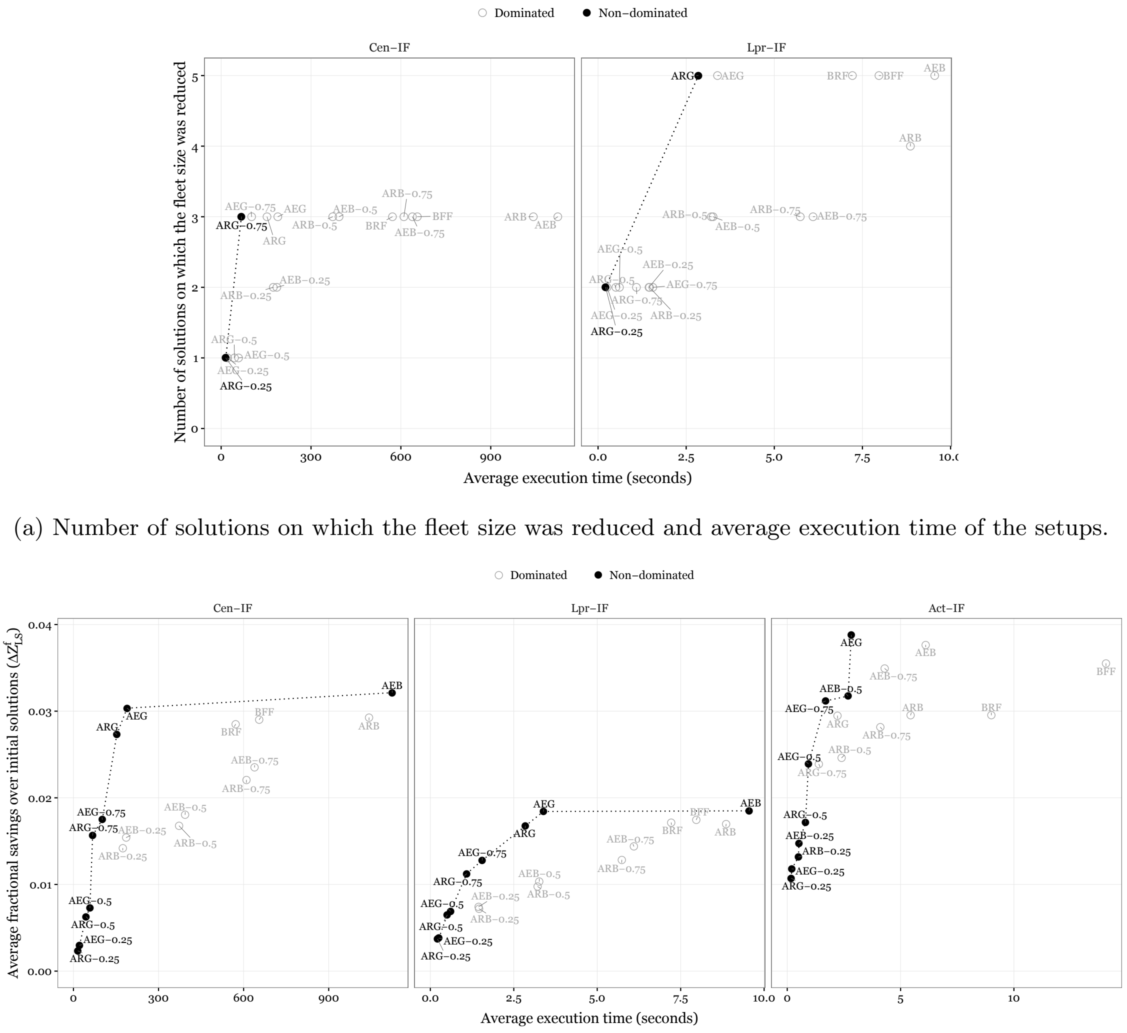
(a) CPU times and trend-lines of the LS setups versus problem instance size $\tau = |\boldsymbol{R}|$.

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

Figure 6.7: Comparison of LS-Accelerated-Reduced-Greedy (LS-ARG), LS-Accelerated-Extended-Greedy (LS-AEG), LS-Accelerated-Reduced-Best (LS-ARB) and LS-Accelerated-Extended-Best (LS-AEB) on waste collection benchmark sets.

from further study. For the analysis, we calculated the average savings per benchmark set obtained by the setups over all the initial solutions, the number of excessive-fleet initial solutions on which they successfully reduced the fleet-size, as well as their average execution times. A setup was then flagged as dominated if another setup produced the same or better quality local optima, but required less computational time to do so.

Results for the analysis are shown in Figure 6.8. The domination of the setups in terms



(a) Number of solutions on which the fleet size was reduced and average execution time of the setups.

(b) Average fractional cost savings and average execution time of the setups.

Figure 6.8: Dominated and non-dominated local search setups in terms of cost savings, fleet reduction and execution time of the setups on waste collection benchmark sets. A full list of acronyms used in the setup can be found in Table 6.3 at the beginning of the setup.

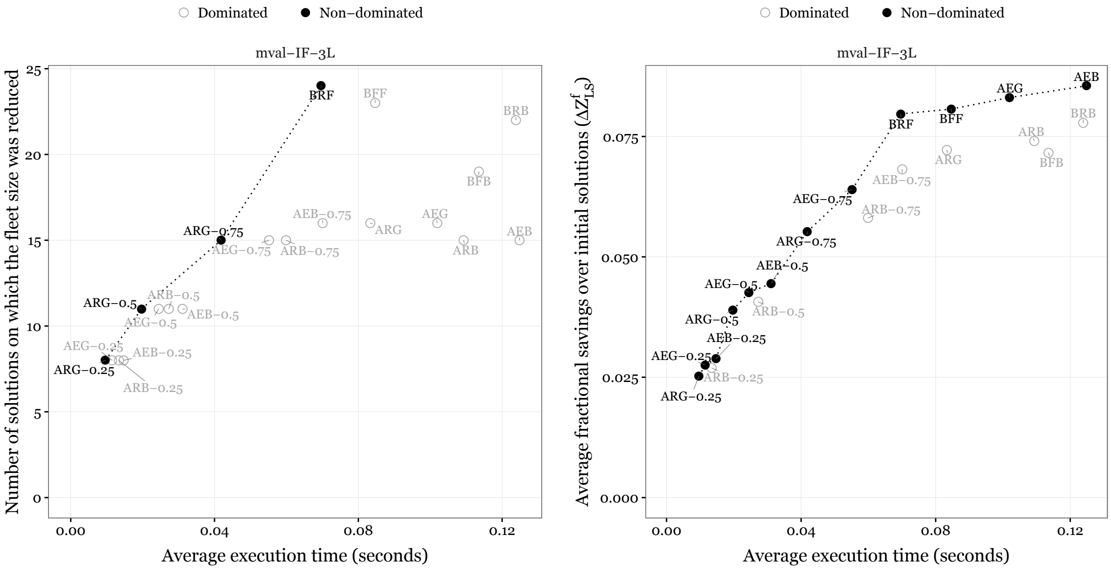
of fleet size reduction is shown in Figure 6.8a. The vehicle fleets could only be reduced on *Cen-IF* and *Lpr-IF*, which is why *Act-IF* does not feature in the figure. On *Cen-IF*, LS-ARG-0.25 and LS-ARG-0.75 were non-dominated as they had the fastest execution times, while producing fleet reductions similar to those of the other setups. On *Lpr-IF*, LS-ARG-0.25 was again non-dominated, as was LS-ARG. The performance of the other setups in conjunction with their cost-reduction dominance meant that they could not be automatically discounted. Two setups, namely LS-ARB-0.25 and LS-AED-0.25, were clearly outperformed on *Cen-IF* by quicker setups that produced better fleet reductions, and required less time to-do so. On *Lpr-IF*, the five setups LS-ARB-0.5, LS-AEB-0.5, LS-ARB-0.75, LS-AEB-0.75 and LS-ARB, were similarly outperformed. As shown in Figure 6.8b, these setups were dominated on cost-savings as well, and could thus be eliminated from further testing. The other setups produced very similar fleet reductions. This necessitated us to rely mostly on cost-savings to eliminate some of them from further testing.

The domination of the setups in terms of average cost savings and computational time is shown in Figure 6.8b. On all three sets, LS-BFF and LS-BRF were always dominated by LS-AEG and subsequently eliminated form further testing. On *Cen-IF* and *Lpr-IF* the LS-AEB setup had the highest execution time, but remained non-dominated since it produced the best local-optima. The *Nearest-Neighbour-Lists* reduced the execution time of the accelerated setups, more so for LS-AEB-0.25 to 0.75 and LS-ARB-0.25 to 0.75, but in most cases the coinciding reduction in solution quality meant that the setups were dominated by LS-ARG-f and LS-AEG-f. Except for LS-ARB-0.25 on *Act-IF*, all the accelerated *best-move* setups with $f \in \{0.25, 0.5, 0.75\}$ were dominated, as well as LS-ARB-0.25 on *Cen-IF* and *Lpr-IF*. As a result, the setups were eliminated from further testing, leaving only LS-AEB from the LS-Accelerated-Best group. LS-ARG and the three LS-ARG-0.25 to 0.75 setups were non-dominated on all three sets. Although a few of the LS-AEG-0.25 to 0.75 setups were dominated on the *Lpr-IF* set, none were eliminated since they were non-dominated on *Act-IF* and *Cen-IF*. In summary, the nine non-dominated setups that were subjected to further testing were LS-AEB, LS-AEG-f and LS-ARG-f, both with $f \in \{0.25, 0.5, 0.75, 1\}$.

Figure 6.8 shows that the non-dominated and dominated setups were fairly consistent between the waste collection sets, although some discrepancies were observed between the *Act-IF* and other two sets. The dominated and non-dominated analysis may therefore give different results when the test sets are substantially different.

Tests on constructive heuristics in Chapter 4 showed that heuristic performance varies between benchmark sets, highlighting the need for tests to be conducted on realistic problem instances. To determine if the same is true for LS improvement procedures, the domination analysis were repeated on the *mval-IF-3L* set. The set has instances similar to the *gdb* and *bccm* sets regularly used for computational tests on CARPs. All sets are discussed in detail in Chapter 2, Section 2.4.2.

Results for the domination analysis on *mval-IF-3L* are shown in Figure 6.9, including results for the four basic LS heuristics from the previous chapter. The differences in the solution quality on this set are much more prominent, whereas the difference in computational times are much smaller. Out of the 102 initial solutions used for the tests, forty-eight had known excessive fleet sizes. The fastest setup, LS-ARG-0.25, reduced the fleets of only seven solutions, whereas LS-BRF reduced the fleet on twenty-solutions. In terms of cost savings, LS-ARG-0.25 had an average savings of 2.5%. The average savings of LS-BRF was close to 8%. With both setups taking, on average, less than 0.075 seconds to improve the initial solutions, it would be difficult to motivate why the quicker, but poorly performing LS-ARG-0.25 setup should be considered for further testing. LS-BRF performed well in reducing costs and the best in reducing fleet sizes, and it was quicker than the accelerated LS-AEG and LS-AEB setups. With its computational time of less than 0.075 seconds, LS-BRF would be a good candidate under short and medium term execution times. Yet, on the waste collection sets, LS-BRF was always dominated by LS-AEG (Figure 6.8). On the large *Cen-IF* instances, LS-BRF took, on average, close to ten



(a) Number of solutions on which the fleet size was reduced and execution time of the setups.

(b) Average fractional cost savings and execution time of the setups.

Figure 6.9: Dominated and non-dominated local search setups in terms of cost savings, fleet reduction and execution time of the setups on the *mval-IF-3L* benchmark set.

minutes to reach local optima, whereas LS-AEG took on average just over two minutes to reach better local optima.

Tests on the waste collection instances show that LS-BRF 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.

Returning to our tests on the waste collection instances. If the LS setups were called to improve the deterministic *Path-Scanning*, *Improved-Merge* and *Efficient-Route-Cluster* solutions, then the best LS heuristic would be LS-AEG-1. The best combined setup would be limited to the constructive heuristics linked with LS-AEG-1 with combined execution times less than predetermined time-limits When additional execution time is available better local optimum solutions may be found through a multi-start application, whereby the LS setups are allowed to improve multiple different initial solutions for the same instance, from the which the best is returned. The question is then whether the quicker setups that terminate at worse local optima will produce better results than the slower better local optima setups under the same total execution time-limits. This brings us to the final computational tests of this chapter.

6.4.3 Multi-start performance evaluation

To compare solution quality of the multi-start and deterministic LS setups over different problem instances the least cost and least vehicle fleet size solutions, found during all our computational tests on the LS setups, were used as a baseline. The solution quality of a local optimum solution, $\boldsymbol{T}^{(t)}$, was then analysed using the cost gap, Z_{gap}, of the solution from the best known cost, calculated as:

$$Z_{\text{gap}} = \frac{Z(\boldsymbol{T}^{(t)}) - Z_{\text{BF}}}{Z_{\text{BF}}}, \tag{6.23}$$

where Z_{BF} is the cost of the best solution found for a problem instance during our LS computational tests.

For vehicle fleet size evaluations, the fleet size gap, K_{gap}, for a heuristic solution was calculated as:

$$K_{\text{gap}} = |\boldsymbol{T}^{(t)}| - K_{\text{BF}}, \tag{6.24}$$

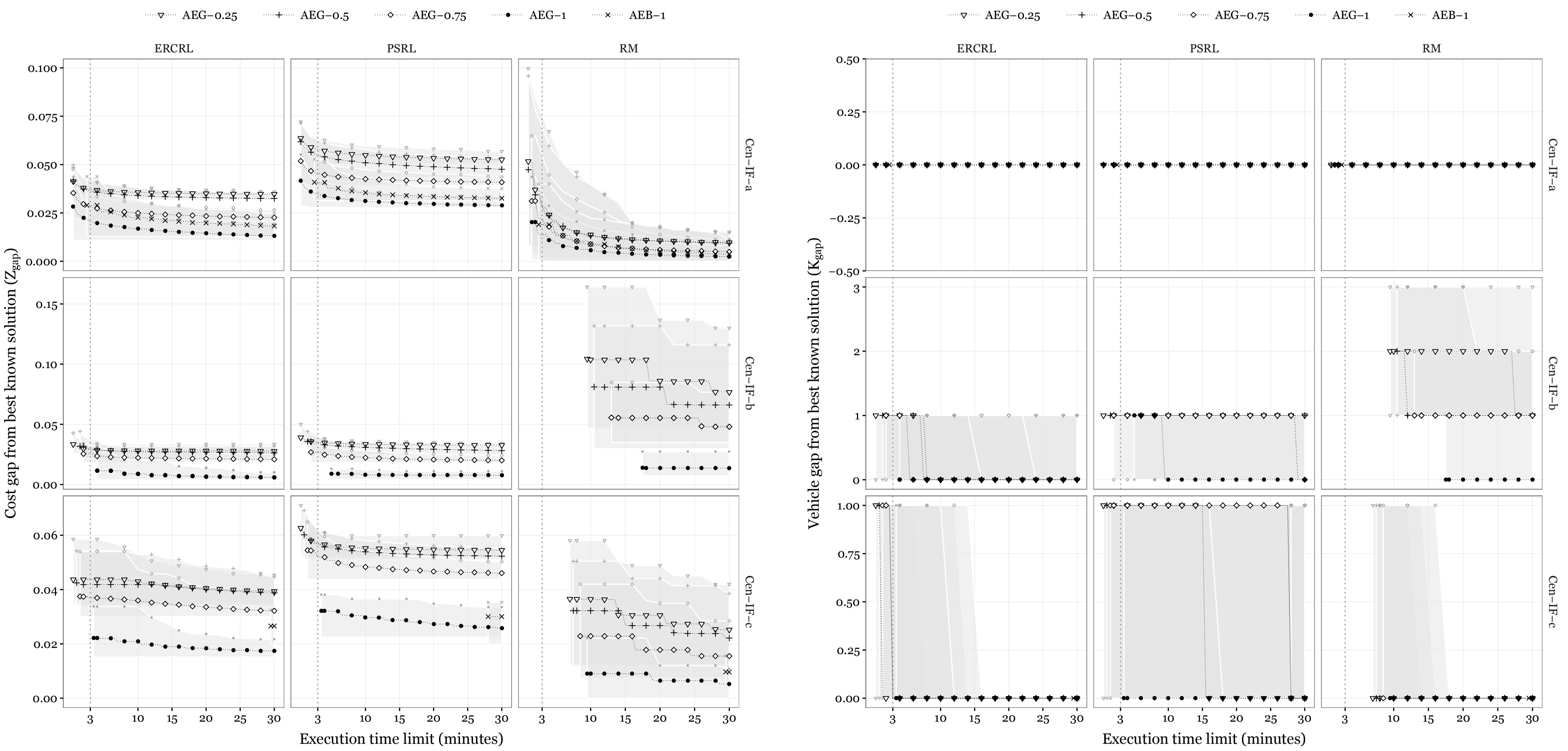
where K_{BF} is the minimum fleet size found for a problem instance during all our LS computational tests.

To evaluate the LS setups under multi-start conditions the following experiments were conducted. First, each LS setup was linked with the following randomised multi-start constructive heuristics: *Efficient-Cluster-Random-Link* (ERCRL), *Path-Scanning-Random-Link* (PSRL) and *Randomised-Merge* (RM). The linked setups were then used to generate and improve up to 1000 solutions under a one-hour time-limit, and the cost and required fleet size for each improved solution was captured. We then calculated the number of solutions that can be generated by the randomised heuristic and improved to local optima by an LS setup within different execution time-limits, ranging from one to thirty-minutes at thirty-second intervals. Each constructed and LS-improved solution represents an independent run. Using the probability mass function of a binomial distribution and taking as input the number of solutions that can be generated within the different time-limits, we then calculated the expected cost gap, $\overline{Z}_{\text{gap}}$, and fleet size gap, $\overline{K}_{\text{gap}}$, of the best improved solution over multiple runs, found within different execution time-limits[2]. For the calculations, minimising K was always set as the primary objective. To evaluate the variance of the setups we calculated the 99th percentile cost gap, $Z_{\text{gap}}^{99^{\text{th}}}$, and fleet size gap, $K_{\text{gap}}^{99^{\text{th}}}$, for different execution time-limits, and used the values as performance upper-bounds. For performance lower bounds we calculated the 1st percentile cost gap, $Z_{\text{gap}}^{1^{\text{st}}}$, and fleet size gap, $K_{\text{gap}}^{1^{\text{st}}}$. Together the performance upper and lower bound values represent the cost and fleet size range of each setup under different execution time-limits. Cost and vehicle gap values are given as an interval $[Z_{\text{gap}}^{1^{\text{st}}}, \overline{Z}_{\text{gap}}, Z_{\text{gap}}^{99^{\text{th}}}]$ and $[K_{\text{gap}}^{1^{\text{st}}}, \overline{K}_{\text{gap}}, K_{\text{gap}}^{99^{\text{th}}}]$. An illustration of the analysis can be found in Section 6.B at the end of the chapter.

Multi-start setups with reduced nearest-neighbour-lists

The first test that we conducted focussed on the impact of the *Nearest-Neighbour-Lists* on the LS-Accelerated-Extended-Greedy-f (LS-AEG-f) and LS-Accelerated-Reduced-Greedy-f (LS-ARG-f) setups. We also compared the setups against LS-Accelerated-Extended-Best (LS-AEB), which had the best improvement capability, but long execution times. Results for the setups linked with the four randomised constructive heuristics on the *Cen-IF* instances are shown in Figure 6.16. Cost gap intervals for the different setups are shown in Figure 6.16a. Due to its long execution time, LS-AEB could not complete a single run on *Cen-IF-b*. It also required in excess of 25 minutes to complete a single run on *Cen-IF-c*, which is too close to the 30 minute medium execution time threshold. The ability of LS-AEG to complete more runs resulted in it dominating LS-AEB. In fact, LS-AEG dominated all the other LS-AEG-f setups with $f \in \{0.25, 0.5, 0.75\}$ over all the constructive heuristics and problem instances. The solution quality of LS-AEG-f decreases with f, meaning that the additional runs that are possible from the very efficient setups do not make up for their worse local optima. These setups may still be useful, especially on the

[2] A more in-depth discussion of the calculation procedure, which is the same procedure used in Chapter 4 to compare the randomised and deterministic constructive heuristics, can be found in Chapter 4, Section 4.4.1, with specific reference to Figure 4.2 and Equations (4.57) and (4.58).



(a) Cost gap interval, $[Z^{1^{st}}_{gap}, \overline{Z}_{gap}, Z^{99^{th}}_{gap}]$, for the best solution found by the LS setups within different execution time-limits.

(b) Fleet size gap interval, $[K^{1^{st}}_{gap}, \overline{K}_{gap}, K^{99^{th}}_{gap}]$, for the best solution found by the LS setups within different execution time-limits.

Figure 6.10: Cost and fleet size gaps of the multi-start LS-Accelerated-Extended-Best-1 (AEB-1) and LS-Accelerated-Extended-Greedy-f (AEG-f) setups linked with *Efficient-Route-Cluster-Random-Link* (ERCRL), *Path-Scanning-Random-Link* (PSRL) and *Randomised-Merge* (RM) constructive heuristics on *Cen-IF* instances.

large *Cen-IF-b* and *Cen-IF-c* instances where LS-AEG requires in excess of 180 seconds to complete a single run. Alternatively, LS-AEG may be terminated prior to reaching a local optimum, and its solution returned.

Fleet size gap intervals for the setups are shown in Figure 6.16b. On *Cen-IF-a*, all setups were able to match best known fleet size. The LS-Accelerated-Reduced-Greedy-f (LS-AEG-f) setups linked with ERCRL were able to match the best known fleet size on *Cen-IF-b*, but required in excess of 180 seconds to be expected to do so. The same was also observed on *Cen-IF-c*, but here at least LS-AEG-0.25 linked with ERCRL was able to reach a zero fleet size gap within 180 seconds. The LS-AEG-1 setup was able to reduce the fleet sizes of the constructive heuristic solutions, but only if sufficient execution times are available.

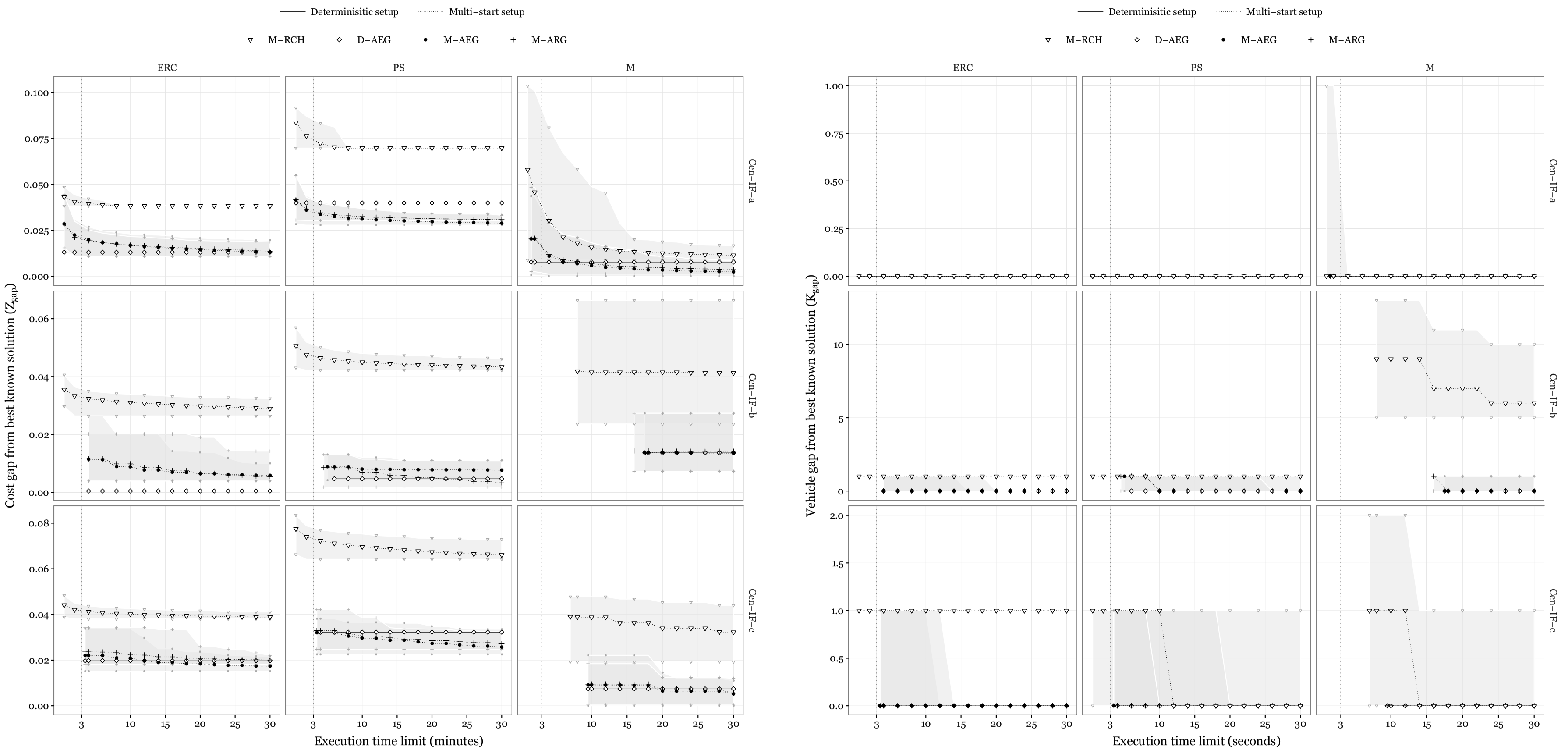
The same results shown in Figure 6.16 were observed for the LS-AEG-f setups on *Cen-IF*, as well for all the setups on *Act-IF* and large *Lpr-IF* instances; the results can be found in Section 6.C at the end of the chapter. On smaller instances, where LS-AEB could execute for multiple runs it performed either the same or worse than LS-ARG. As a result, it was eliminated from further testing. On all the waste collection instances, the LS-AEG-1 and LS-ARG-1 setups performed the best under time-limits, and the setups with the lowest f values performed the worst. As a result, the LS-AEG-0.25 to LS-AEG-0.75 and LS-ARG-0.25 to LS-ARG-0.75 setups were also eliminated from further testing.

Accelerated LS under short and medium time-limits

For our final analysis we compared the multi-start LS-ARG and LS-AEG setups, annotated M-ARG and M-AEG, linked with randomised constructive heuristics, against the pure multi-start randomised constructive heuristics, annotated as M-RCH, as well as LS-AEG linked with the deterministic constructive heuristics, annotated as D-AEG. The aim of the analysis was to find the best constructive heuristic and LS setup under short and medium execution time-limits.

Summary results for the setups on the *Cen-IF* instances are shown in Figure 6.11. As shown in Figure 6.11a, the cost gaps of the deterministic and multi-start LS setups were less than the pure M-RCH setups, which confirms that our LS implementations are effective in improving initial solutions of constructive heuristics, and efficient enough to do so within short time-limits. As shown in Figure 6.11b, the LS setups were also effective in reducing the fleet sizes of initial solutions. In four of the nine cases, M-RCH produced excess fleet solutions even when allowed thirty minutes of execution time. In all cases, the multi-start LS setups reduced the fleet to its best-known size, but they did require more than three minutes of execution-time to do so.

Returning to Figure 6.11a, the cost gap intervals of the multi-start LS-ARG and LS-AEG setups were very close, with the dominance of any one setup over the other being dependent on the problem instance solved and the constructive heuristic used to supply initial solutions. The deterministic setup performed surprisingly well, especially when linked with ERC, even outperforming the multi-start setups on the *Cen-IF-b* instance. A downside of the multi-start setups is that they have a wide cost-range. When more execution time is available, the expected cost of the multi-start setups approaches their lower-bound cost-values, but their cost ranges remain wide on the large instances. On these instances, where only a few LS runs are possible, the deterministic LS-AEG setup linked with ERC is the best option given its consistent and good performance. On the very large *Cen-IF-b* and *Cen-IF-c* instances, with more than 5000 required arcs and edges, none of the LS setups could complete a single run within 180 seconds. When limited execution time is available one can thus rely exclusively on the constructive heuristics, or consider



(a) Cost gap interval, $[Z^{1^{st}}_{gap}, \overline{Z}_{gap}, Z^{99^{th}}_{gap}]$, for the best solution found by the setups within different execution time-limits.

(b) Fleet size gap interval, $[K^{1^{st}}_{gap}, \overline{K}_{gap}, K^{99^{th}}_{gap}]$, for the best solution found by the setups within different execution time-limits.

Figure 6.11: Cost and fleet size gaps on *Cen-IF* instances of the multi-start (M) and deterministic (D) setups of LS-Accelerated-Reduced-Greedy (ARG) and LS-Accelerated-Extended-Greedy (AEG) linked with *Efficient-Route-Cluster* (ERC), *Path-Scanning* (PS) and *Merge* (M) constructive heuristics, as well as the multi-start randomised constructive heuristic (M-RCH) setups on their own.

using an LS-ARG-f or LS-AEG-f setup, with f set sufficiently low. Alternatively, and the option that we used in this thesis, the LS-ARG and LS-AEG setups can be terminated prior to reaching local optima.

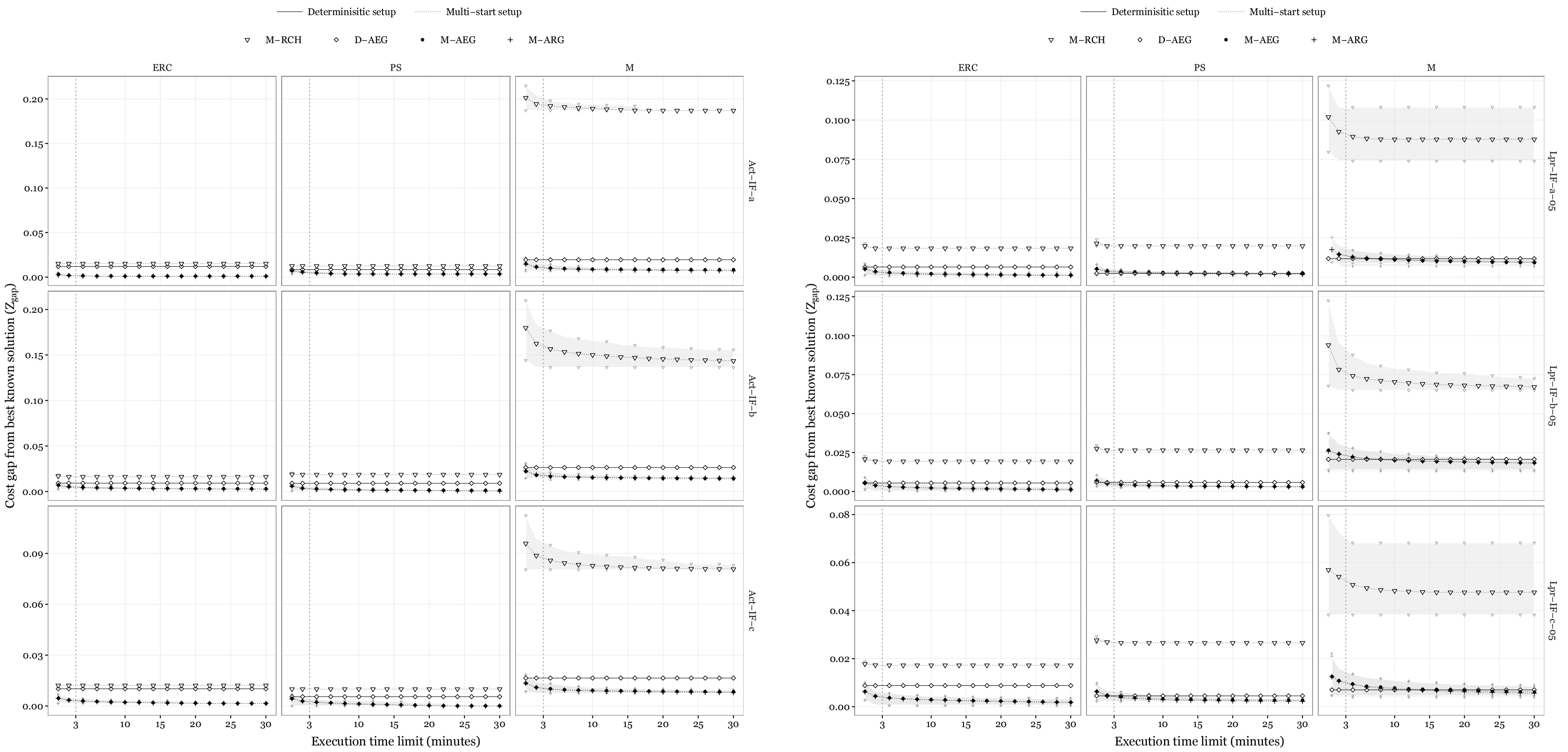
Cost-gap results for the setups on the *Act-IF* set and large *Lpr-IF* type 5 instances are shown in Figure 6.12. As shown in Figure 6.12a, RM performed extremely poorly, with cost-gaps close to 20% on *Act-IF-a*. The multi-start LS setups were able to significantly improve the RM solutions, bringing it on par with the solutions of ERCRL and PSRL. The best performing setups were ERC and PS linked with LS-AEG and LS-ARG, which performed almost identically. With their extra runs they also outperformed the deterministic LS-AEG setup. The same results were observed on the large *Lpr-IF* instances (Figure 6.12b). On all instances, the LS setups were able to produce good quality solutions within 180 seconds. The setups also produced solutions with fleet sizes equal to best-known values.

To compare the different LS setups under short and medium execution time-limits, the expected cost gaps and fleet-sizes of the setups were measured at three and thirty minute execution time-limits. The values were then compared against the MRCH setups at the same time-limits, as well as the pure Deterministic Constructive Heuristics (DCHs). Average results over the *Act-IF*, *Cen-IF*, *Lpr-IF*, as well as the smaller sized *mval-IF-3L* instances are shown in Table 6.5.

Over all the instances, the best performing LS setups under medium execution time-limits were the multi-start LS-AEG and LS-ARG setups linked with ERCRL. In comparison to the MRCH setups, the multi-start LS setups produced significantly better solutions, with their average cost-gaps always being less than 2.1%, regardless of the constructive heuristics with which they were linked. Cost gaps of the MRCH setups were in excess of 10% in some cases, thus confirming the benefit of developing and implementing LS heuristics for the MCARPTIF. LS made the biggest impact on the *mval-IF-3L* instances, but the results have to be interpreted with caution, since the instances are not representative of realistic waste collection, in terms of size and network configuration.

Where an execution time-limit of three-minutes was enforced, the best performing setups were again LS-AEG and LS-ARG linked with ERCRL. The LS setups produced significantly better solutions than the MRCH setups, but only on the *Act-IF*, *Lpr-IF* and *mval-IF-3L* instances. On the large *Cen-IF-a* and *Cen-IF-b* instances, the deterministic and multi-start LS-AEG and LS-ARG setups terminated prior to reaching local optimum. The setups linked with IM and RM could not be used at all due the time required to construct a single initial solution exceeding three minutes. By terminating before a local optimum could be reached the differences in performance between the LS-AEG and LS-ARG setups and MRCH setups were less prominent. This shows that more execution time is required for LS to be truly effective under short execution time-limits on realistically sized instances. LS-AEG linked with ERC performed the best over all three waste collection sets, and it has the benefit of being completely deterministic, whereas the multi-start setups have a wide cost ranges when limited to a few runs.

Table 6.6 shows the total number of instances per benchmark set for which the setups produced solutions with excess fleet sizes. The best performing LS setups were again the multi-start LS-AEG and LS-ARG setups linked with ERCRL. When allowed 30 minutes of execution time, the two setups matched the best known fleet sizes on 53 out of 55 instances. The deterministic LS-AEG setup linked with ERC again performed well on the waste collection sets, producing an excess fleet size on only one instance. The setup also performed well under short time-limits, though it did produce excess fleet solutions on two of the three *Cen-IF* instances. This can be mitigated by relaxing the time-limits from



(a) Cost gap interval, $[Z^{1\text{st}}_{\text{gap}}, \overline{Z}_{\text{gap}}, Z^{99\text{th}}_{\text{gap}}]$, for the best solution found by the setups within different execution time-limits on *Act-IF*.

(b) Cost gap interval, $[Z^{1\text{st}}_{\text{gap}}, \overline{Z}_{\text{gap}}, Z^{99\text{th}}_{\text{gap}}]$, for the best solution found by the setups within different execution time-limits on *Lpr-IF*.

Figure 6.12: Cost and fleet size gaps on *Act-IF* and *Cen-IF* instances of the multi-start (M) and deterministic (D) setups of LS-Accelerated-Reduced-Greedy (ARG) and LS-Accelerated-Extended-Greedy (AEG) linked with *Efficient-Route-Cluster* (ERC), *Path-Scanning* (PS) and *Merge* (M) constructive heuristics, as well as the multi-start randomised constructive heuristic (M-RCH) setups on their own.

t

Table 6.5: Averages for the expected percentage cost gaps, $\overline{Z}_{\text{gap}}\,\%$, of heuristic setups under short-term and medium-term and planning execution time-limits of three and thirty minutes, respectively.

		$\overline{Z}_{\text{gap}}\,\%$ at 3 minutes					$\overline{Z}_{\text{gap}}\,\%$ at 30 minutes				
Set	Construct	DCH	MRCH	DAEG	MAEG	MARG	DCH	MRCH	DAEG	MAEG	MARG
Act-IF	ERC	2.2	1.5	1.0	0.3	0.3	2.2	1.5	1.0	0.2	0.2
	IM	13.9	14.6	2.1	1.2	1.3	13.9	13.7	2.1	1.0	1.0
	PS	2.0	1.4	0.8	0.4	0.3	2.0	1.4	0.8	0.2	0.1
	Mean	6.0	5.8	1.3	0.6	0.6	6.0	5.5	1.3	0.5	0.4
Cen-IF	ERC	2.8	3.8	3.0*	3.0*	3.0*	2.8	3.5	1.1	1.2	1.3
	IM	-	-	-	-	-	2.0	2.8	1.0	0.7	0.8
	PS	7.1	6.5	4.5*	4.1*	4.1*	7.1	6.0	2.6	2.0	2.0
	Mean	5.0	5.1	3.8	3.6	3.6	3.9	4.1	1.5	1.3	1.4
Lpr-IF	ERC	2.0	1.3	0.8	0.2	0.2	2.0	1.3	0.8	0.1	0.1
	IM	3.8	4.3	1.1	0.9	0.8	3.8	3.9	1.1	0.6	0.6
	PS	2.6	1.9	0.8	0.3	0.3	2.6	1.9	0.8	0.2	0.3
	Mean	2.8	2.5	0.9	0.5	0.5	2.8	2.4	0.9	0.3	0.3
mval-IF-3L	ERC	16.4	10.9	8.3	0.8	0.8	16.4	10.9	8.3	0.7	0.8
	IM	21.3	9.2	10.2	1.4	1.6	21.3	8.8	10.2	1.1	1.2
	PS	16.5	8.3	8.4	1.6	1.8	16.5	8.3	8.4	1.5	1.7
	Mean	18.0	9.4	9.0	1.3	1.4	18.0	9.3	9.0	1.1	1.2
Global	ERC	5.8	4.4	3.3	0.4	0.5	5.8	4.3	2.8	0.5	0.6
	IM	-	-	-	-	-	10.3	7.3	3.6	0.8	0.9
	PS	7.1	4.5	3.6	0.8	0.8	7.1	4.4	3.1	1.0	1.0
	Mean	6.5	4.4	3.4	0.6	0.6	7.7	5.3	3.2	0.8	0.8

DCH: Deterministic Constructive Heuristic; MRCH: Multi-start Randomised Constructive Heuristic; DAEG: Deterministic LS-AEG; MAEG: Mulit-start LS-AEG; MARG: Multi-start LS-ARG. *In cases where the execution time of the LS-AEG and LS-ARG setups were too long to complete a single run, the heuristic was terminated prior to reaching local optima.

Table 6.6: Number of instances on which the of heuristic setups failed to produce solutions with the known minimum fleet size.

		3 minute execution time-limits					30m execution time-limits				
Set	Construct	DCH	MRCH	DAEG	MAEG	MARG	DCH	MRCH	DAEG	MAEG	MARG
Act-IF	ERC	0	0	0	0	0	0	0	0	0	0
	IM	2	0	0	0	0	2	0	0	0	0
	PS	0	0	0	0	0	0	0	0	0	0
	Total	2	0	0	0	0	2	0	0	0	0
Cen-IF	ERC	3	2	2*	1*	2*	3	2	0	0	0
	IM	-	-	-	-	-	2	1	0	0	0
	PS	2	2	2*	2*	2*	2	1	0	0	0
	Total	5	4	4	3	4	7	4	0	0	0
Lpr-IF	ERC	9	1	1	0	0	9	1	1	0	0
	IM	8	6	1	0	0	8	5	1	0	0
	PS	2	1	1	0	0	2	1	1	0	0
	Total	19	8	3	0	0	19	7	3	0	0
mval-IF-3L	ERC	16	18	12	2	2	16	18	12	2	2
	IM	24	12	19	5	5	24	12	19	5	5
	PS	15	7	12	4	4	15	7	12	4	4
	Total	55	37	43	11	11	55	37	43	11	11
Global total	ERC	28	21	15	3	4	28	21	13	2	2
	IM	36	18	20	5	5	36	18	20	5	5
	PS	19	10	15	6	6	19	9	13	4	4
	Total	83	49	50	14	15	83	48	46	11	11

DCH: Deterministic Constructive Heuristic; MRCH: Multi-start Randomised Constructive Heuristic; DAEG: Deterministic LS-AEG; MAEG: Multi-start LS-AEG; MARG: Multi-start LS-ARG. *In cases where the execution time of the LS-AEG and LS-ARG setups were too long to complete a single run, the heuristic was terminated prior to reaching local optima.

three to four minutes, thereby allowing the setups to reach local optima.

Results for the setups on the MCARPTIF instances show that the best setup under short and medium time-limits is LS-AEG linked with ERCRL. LS-AEG linked with ERC also performed well and it has the benefit of being completely deterministic. The choice between using randomised multi-start or deterministic setups need not be mutually exclusive. Our recommended option would be to always use LS-AEG to improve the initial solution from ERC for a first run. Thereafter, LS-AEG can be used to improve multiple initial solutions generated by ERCRL until an imposed execution time-limit is reached. This will significantly improve the worst-case performance of the setup, and enable it to produce consistent results, even on large instances where only a few randomised runs, if any, can be completed.

6.5 Conclusion

In this section we evaluated accelerated LS setups for the MCARPTIF. 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. 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 *Nearest-Neighbour-Lists* mechanism 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. Although not considered in this thesis, the implementations of both mechanisms 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.

In the previous chapter we showed that basic LS setups are too slow to practically deal with large MCARPTIF instances. On instances with more than 1000 required arcs and edges, the basic setups took between fifteen minutes and three hours to improve a single solution. The accelerated setups, developed in this chapter, took at most four minutes to improve the same solutions, with the most efficient version taking less than 60 seconds. Our 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 the next chapter the efficient LS-Accelerated-Extended-Greedy heuristic is further extended into a Tabu Search metaheuristic, and its performance compared against pure LS on large waste collection problem sets. The metaheuristic is also tested on MCARP and CARPTIF benchmark sets and compared against existing metaheuristics for the problems. Lastly, to evaluate the practical limits of our heuristics, further tests are performed on a huge MCARPTIF instance with 6280 required arcs and edges.

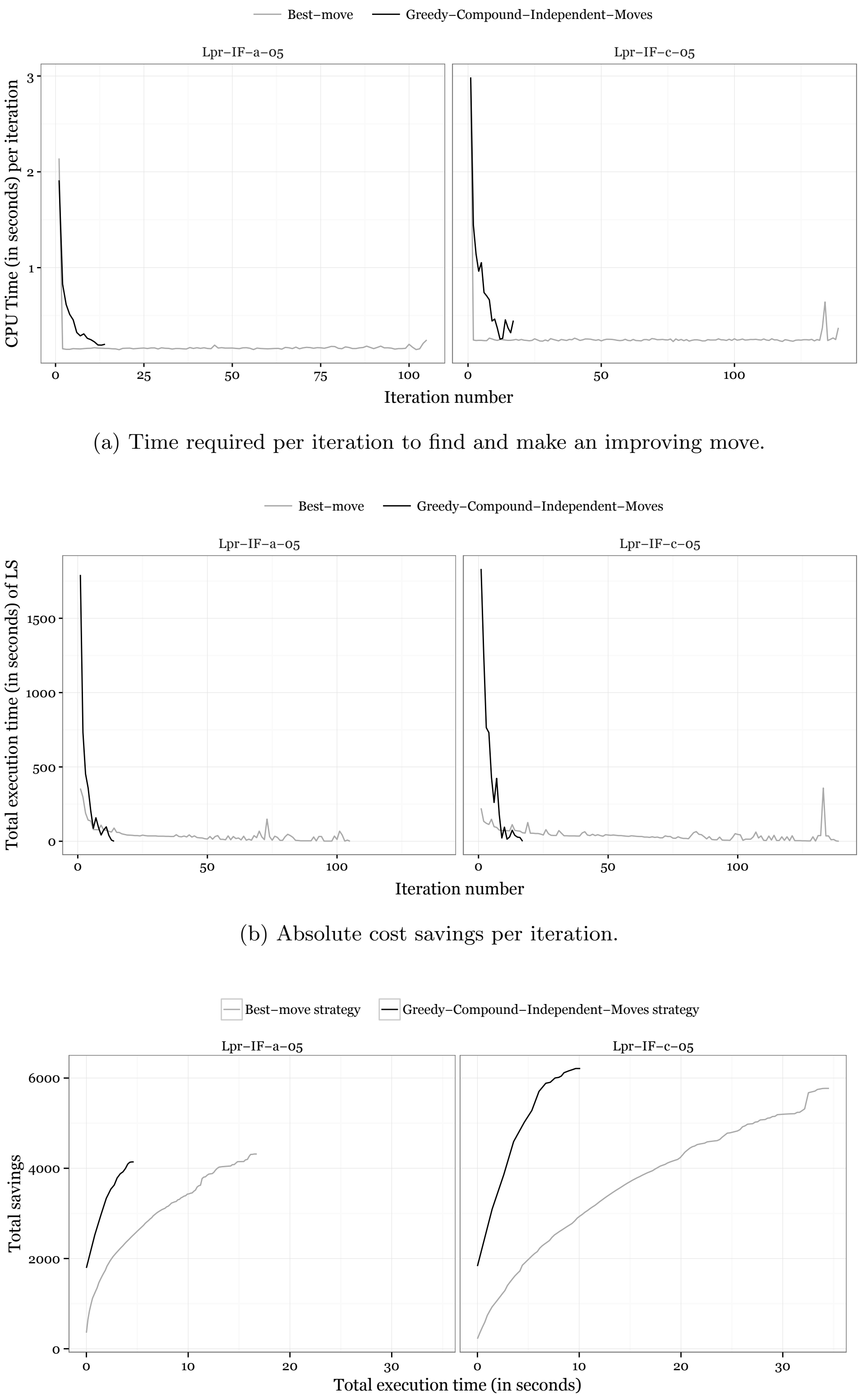
Chapter appendix

6.A Accelerated best-move versus accelerated compounded moves

Figure 6.13 illustrates the difference between only using *Static-Move-Descriptors* and when combining it with *Greedy-Compound-Independent-Moves*. For both setups, in the first iteration the entire move neighbourhood is scanned to populate the move savings list, which is why the first iteration takes the longest to complete, as shown inFigure 6.13a. In the following iterations, the computational times of the setups are significantly less. With the *best-move* strategy, after making the best move, LS only has to update the affected move descriptors and reorder the savings list. *Greedy-Compound-Independent-Moves* makes multiple moves in each iteration. As a result, its computational time per iteration is higher since it has to update the descriptors of more than one move. As it approaches the local optimum, there are less improving moves available and its per iteration time converges to that of *best-move*. It reaches the local optima after only a few iterations, whereas *best-move* takes over a hundred-iterations. As shown in Figure 6.13c, with its multiple moves, the savings obtained in each iteration of *Greedy-Compound-Independent-Moves* is high in the first few iterations, but reduces as it approaches local optima. In comparison, the savings of *best-move* is less per iteration since it only makes a single move. As shown in Figure 6.13c, by making multiple moves per iteration the total execution time of *Greedy-Compound-Independent-Moves* is less than that of *best-move*. On the *Lpr-IF-a-05* instance *best-move* reached a better local optimum whereas *Greedy-Compound-Independent-Moves* reached a better local optimum on *Lpr-IF-c-05*.

6.B Multi-start analysis

An illustration of the multi-start analysis is shown in Figure 6.14, which focusses on *Path-Scanning-Random-Link* (PSRL) linked with LS-AEG-0.25 and LS-AEG on the *Cen-IF-b* problem instance. On average, the time required by PSRL to generate an initial solution is about 5 seconds. LS-AEG-0.25 and LS-AEG then took, on average, 25 and 230 seconds, respectively, to improve the solution to local optima. As shown in Figure 6.14a, PSRL linked with the much quicker LS-AEG-0.25 setup can generate and improve up-to sixty solutions within thirty minutes, whereas PSRL and LS-AEG can only generate and improve up-to six solutions and needs at least 235 seconds to complete one run. Figure 6.14b shows the expected cost of the best improved solution found within different execution time-limits by the setups, which is based on the number of runs that can be completed by the setups within the time-limits. It also shows the cost-range for each setup, ranging from the 1st to 99th percentile, at different execution time-limits. Even though LS-AEG-0.25 can perform more runs than LS-AEG under equal time-limits, the ability of LS-AEG to find better local optima gives it a better expected performance when

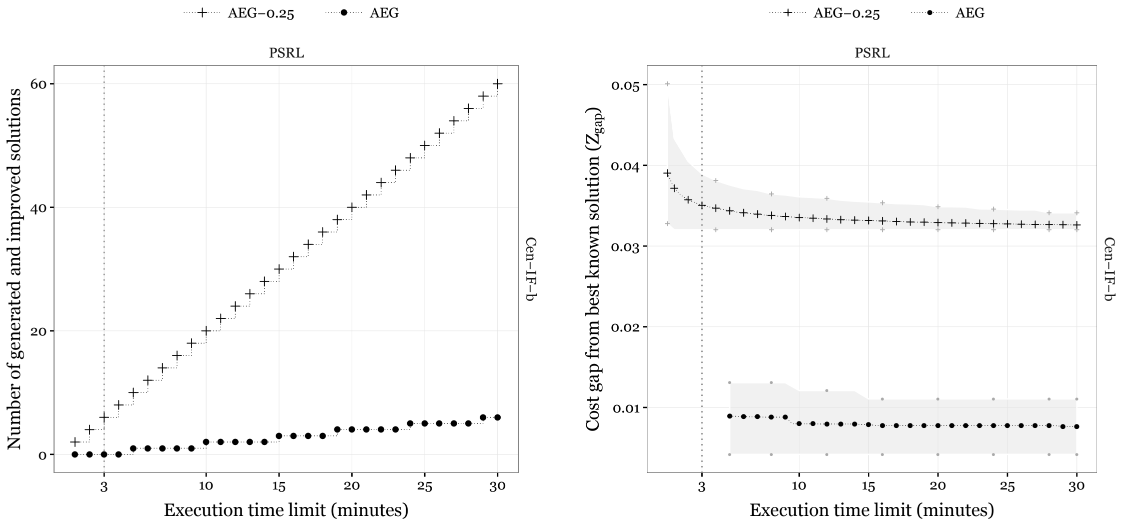


(a) Time required per iteration to find and make an improving move.

(b) Absolute cost savings per iteration.

(c) Total savings over the execution-time of LS.

Figure 6.13: Illustration of the difference between LS only using *Static-Move-Descriptors* and using it in combination with *Greedy-Compound-Independent-Moves*. The LS setups only employed the *relocate* move operator and initial solution were generated using *Path-Scanning*.



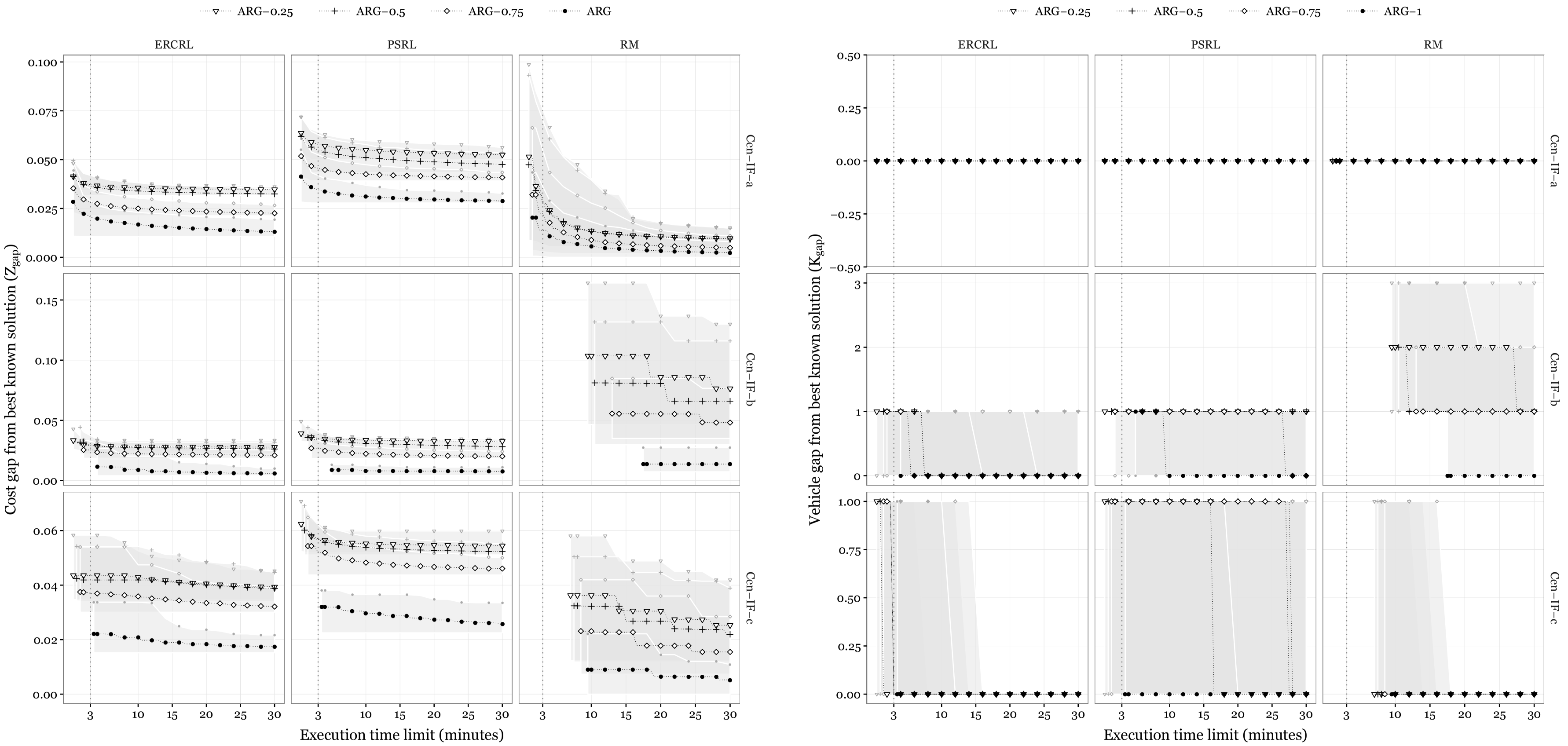
(a) Number of solutions that can be generated with *Path-Scanning-Random-Link* and improved with the LS setups within different execution time-limits.

(b) Cost gap interval, $[Z_{\text{gap}}^{1^{\text{st}}}, \overline{Z}_{\text{gap}}, Z_{\text{gap}}^{99^{\text{th}}}]$, for the best solution found by the LS setups within different execution time-limits.

Figure 6.14: Illustration of the multi-start analysis on LS-AEG-0.25 and LS-AEG setups with *Path-Scanning-Random-Link*.

more than 235 execution-seconds are available. With its few runs, LS-AEG has a wider cost-range, but its upper-bound cost-value is still better than the lower-bound cost-value of LS-AEG-0.25. When taking the cost-range and expected performance of the setups into consideration, LS-AEG is the better setup for medium term planning. Recall that for short-term planning we imposed a three minute time-limit, in which case LS-AEG cannot be used. LS-AEG-0.25, with its thirty second run-time, can be used for this purpose.

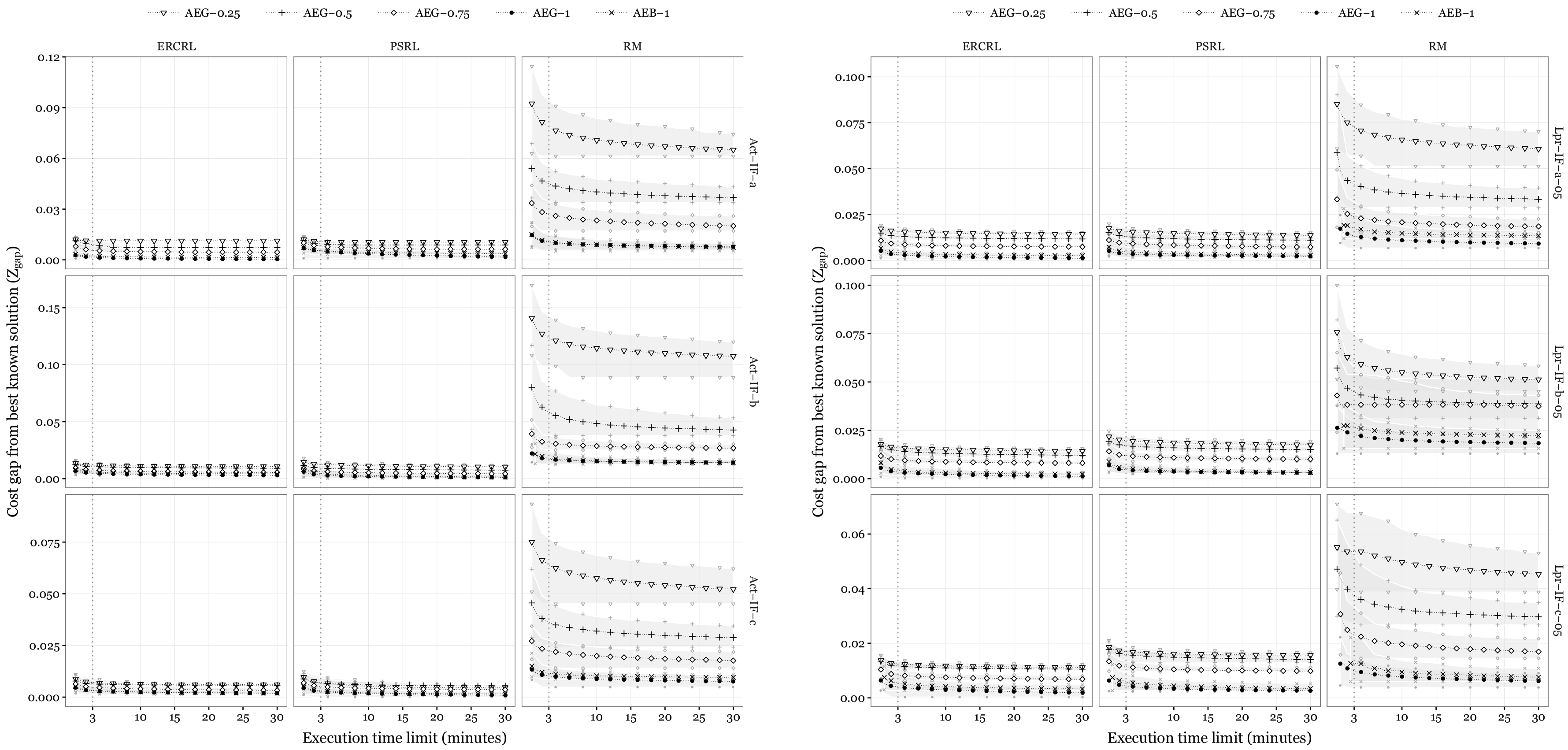
6.C Results for multi-start LS setups with nearest-neighbour-lists



(a) Cost gap interval, $[Z^{1^{\text{st}}}_{\text{gap}}, \overline{Z}_{\text{gap}}, Z^{99^{\text{th}}}_{\text{gap}}]$, for the best solution found by the LS setups within different execution time-limits.

(b) Fleet size gap interval, $[K^{1^{\text{st}}}_{\text{gap}}, \overline{K}_{\text{gap}}, K^{99^{\text{th}}}_{\text{gap}}]$, for the best solution found by the LS setups within different execution time-limits.

Figure 6.15: Cost and fleet size gaps of the multi-start LS-Accelerated-Extended-Best-1 (AEB-1) and LS-Accelerated-Reduced-Greedy-f (AEG-f) setups under execution time-limits on *Cen-IF* instances.



(a) Cost gap interval, $[Z_{\text{gap}}^{1\text{st}}, \overline{Z}_{\text{gap}}, Z_{\text{gap}}^{99\text{th}}]$, for the best solution found by the LS setups within different execution time-limits on *Act-IF* instances.

(b) Cost gap interval, $[Z_{\text{gap}}^{1\text{st}}, \overline{Z}_{\text{gap}}, Z_{\text{gap}}^{99\text{th}}]$, for the best solution found by the LS setups within different execution time-limits on *Lpr-IF* instances.

Figure 6.16: Cost gaps of the multi-start LS-Accelerated-Extended-Best-1 (AEB-1) and LS-Accelerated-Extended-Greedy-f (AEG-f) setups linked with *Efficient-Route-Cluster-Random-Link* (ERCRL), *Path-Scanning-Random-Link* (PSRL) and *Randomised-Merge* (RM) constructive heuristics on *Act-IF* and *Lpr-IF* instances.

Chapter 7

An accelerated tabu search metaheuristic

In this chapter we extend the efficient Local Search (LS) heuristics into a Tabu Search (TS) metaheuristic for the Mixed Capacitated Arc Routing Problem under Time Restrictions with Intermediate Facilities (MCARPTIF). Computational tests are performed on large waste collection instances on which the metaheuristic proved to be effective, outperforming the constructive heuristic and LS setups under short, medium and long execution time-limits. We also test our TS on available Mixed Capacitated Arc Routing Problem (MCARP) and Capacitated Arc Routing Problem under Time Restrictions with Intermediate Facilities (CARPTIF) benchmark sets, and compare the solution quality of our methods against available lower-bound values, and against two existing metaheuristics. On large MCARP instances our TS again proved to be effective, by finding new best solutions for the instances, and efficient, as it found those new best solutions in under 5 minutes. This shows that on large instances there is much room for improvement over the existing methods. A final test was conducted on a huge instance with 6280 required arcs and edges. Two of our constructive heuristics were able to quickly generate starting solutions, whereafter our efficient LS setup improved the solutions to local optima within thirty-minutes. Thereafter TS further improved the solutions within a 24-hour execution time-limit.

7.1 Introduction

In the previous chapter we developed efficient LS heuristics for the MCARPTIF, which on their own significantly improved the initial solutions generated by constructive heuristics. A drawback of these methods is that they terminate at local optima. Single solution metaheuristics, such as Variable Neighbourhood Research, Tabu Search and Guided Local Search, have been designed specifically to overcome this issue [80]. They allow LS to escape local optima and intelligently guide the search through the solution space towards unexplored regions, with the aim of eventually reaching a global optimum.

In this chapter we develop a TS heuristic for MCARPTIF by extending the efficient local search mechanisms developed in the previous chapter. The aim of this chapter is not to come up with the best possible metaheuristic for the MCARPTIF, one that will be difficult to beat in future studies on our benchmark sets. The drawback of this aim, referred to as *competitive testing*, is well documented by Hooker [47], and more recently by Sörensen [79]. One of their main criticisms being that it does not advance practical relevance of research on heuristics and metaheuristics. In line with the two authors' advocation, our aim is instead to improve the general understanding of a key ingredient of metaheuristics, namely LS. The purpose of this chapter is to determine if our research on efficient LS procedures can be easily extended to develop an efficient metaheuristic. If successful, it opens the door for existing Capacitated Arc Routing Problem (CARP) metaheuristics that rely on LS to be improved, thereby bringing CARP research one step closer to the development of efficient solution methods for the huge waste collection instances found in practice.

Thus far the quality of the solutions produced by our heuristics has been internally evaluated. Although our tests indicate that certain heuristics are better than others, there is a risk that all our heuristics are still inadequate for waste collection planning, with the "good" heuristics actually performing poorly, and the other methods performing very poorly. To better evaluate the quality of our solution methods we performed computational tests on the MCARP. The MCARPTIF can be reduced to MCARP, thereby making it possible to use our algorithms as-is on the problem. Tests were performed on the available MCARP benchmark sets, and our solution methods were critically evaluated using lower-bound values for the instances, reported in [6, 40], and against the Memetic Algorithm of Belenguer et al. [6]. Tests were also conducted on existing CARPTIF instances, and the performance of our heuristics was evaluated against the Variable Neighbourhood Search metaheuristic of Polacek et al. [68]

As a final evaluation of our heuristics we performed computational tests on a huge waste collection instance with 6280 required arcs and edges. The largest instance in literature used for testing, prior to our *Cen-IF* set, is the *Lpr-c-05* instance with only 803 required arcs and edges. The purpose of the last test was to establish application boundaries for our heuristics and to identify specific components that should be improved, in future work, to allow the algorithms to more efficiently deal with similarly sized instances.

7.2 Tabu search for the MCARPTIF

LS starts with an initial solution and progressively moves to an improving neighbour solution. When none are available the search is said to be stuck in a local optimum and it terminates. One way to escape local optima is to allow LS to move to non-improving solutions with the hope that these moves will eventually lead to a better local optima, or even the global optimum. If left unchecked this may result in cycling whereby the same few solutions are visited over and over again until the search is terminated. To illustrate, assume LS is at a local optimum and it always makes the best available move. At its current position the best available move is a non-improving move out of the local optimum. After making this move, its reversal now represents an improving move since it returns the search back towards the local optimum, which is better than the current solution. The non-improving move is then duly reversed, whereafter it is again made since it is still the best move out-of the local optimum. The search will then cycle between making and reversing the same non-improving move. A popular way of avoiding this is to enable LS to "remember" solutions that it recently visited, and consider them as *tabu*. LS heuristics that are extended via this mechanism are classified as Tabu Search metaheuristics, originally developed by Glover [36], and they represent some of the most widely used metaheuristic methods [80]. It is also the metaheuristic that we chose to apply to the MCARPTIF.

Since TS is an extension of LS, the advances that we made in the previous chapter on efficient LS heuristics automatically caries over to TS implementations. We do, however, note that TS is one of many LS based metaheuristics that can be applied to CARPs, all which could potentially benefit from our work on LS. We leave their development and evaluation against our TS for future work.

7.2.1 Basic instruments

Algorithm 7.1 shows the high-level framework of our TS implementation for the MCARPTIF. In its simplest form it stores information of recently applied moves in a tabu-list, $\boldsymbol{\beta}$, and then uses the information to identify moves that will result in the search returning to a previously visited solution. Such moves are made tabu and are not allowed to be implemented. In so doing, TS avoids cycling. Populating the tabu-list with information on all the visited solutions is not practical for efficiency reasons. Instead, only partial move information is stored, and only for a limited period. After a certain number of iterations has passed since a move was made, referred to as the *tabu tenure* and denoted by γ, the information associated with the move is deleted from the list.

Algorithm 7.1: *Tabu-Search*

```
Input  : An initial solution $\boldsymbol{T}^{(0)} \in \boldsymbol{X}$, execution time-limit $\text{time}_{\text{lim}}$, and a tabu tenure $\gamma$.
Output: Incumbent solution $\boldsymbol{T}^*$.
$t = 0$;
$\boldsymbol{\beta} = []$ // the tabu-list is initially empty //;
$\boldsymbol{T}^* = \boldsymbol{T}^{(0)}$ // the initial solution is set as the incumbent //;
repeat
    Using Local Search move operators, scan the move-neighbourhood of $\boldsymbol{T}^{(t)}$ and find the best
      feasible move, and using the tabu-list $\boldsymbol{\beta}$ find the best non-tabu feasible move.;
    Implement the best feasible move and let $\boldsymbol{T}'$ be the result;
    if $\boldsymbol{T}'$ is a better solution than $\boldsymbol{T}^*$ then // either in cost or fleet size //
        $\boldsymbol{T}^* = \boldsymbol{T}'$ // the improved solution is set as the new incumbent //;
        $\boldsymbol{T}^{(t+1)} = \boldsymbol{T}^{(t)}$;
    else
        Implement the best non-tabu feasible move and let $\boldsymbol{T}^{(T+1)}$ be the result // if none could
          be found the search terminates //;
        Add the necessary information of the best non-tabu move to $\boldsymbol{\beta}$;
    $t = t + 1$;
    Remove move information from $\boldsymbol{\beta}$ that have been in the tabu-list for more than $\gamma$ iterations;
until the execution time-limit, $\text{time}_{\text{lim}}$, is reached or a feasible move could not be made;
return ($\boldsymbol{T}^*$)
```

During its execution TS keeps track of and updates the best solution found so far, referred to as the *incumbent* solution. Tabu criteria may be overwritten under specific conditions, referred to as *aspiration criteria*. The most commonly used criterion is to allow a tabu move if it results in a new incumbent solution. Ideally TS would stop when it reaches the global optimum, but given the nature of heuristics there is no way of knowing when this actually occurs, unless explicit information is available on the optimal solution cost. Different stopping criteria can instead be used to terminate the search, such as stoping the search after a certain number of moves have been made, or if the algorithm has failed to find a new incumbent solution in the last t moves. Alternatively, and consistent with all our previous computational tests, the search can terminate at a specified execution time-limit.

More advanced mechanisms have been developed to improve the performance of TS, such as medium-term memory to identify common elements in high-quality solutions, and long-term memory to guide the search to unexplored regions of the solution space [80].

With our aim to demonstrate the usefulness of our efficient LS heuristics within a metaheuristic, we chose to limit our implementation to its bare-minimum components, and purposefully avoided adding mechanisms that require user-set parameters. We also chose to reuse our existing LS components wherever possible.

7.2.2 Application of efficient local search components within Tabu Search

The best performing LS setup, developed in the previous chapter, was found to be LS linked with *Static-Move-Descriptors* and *Greedy-Compound-Independent-Moves*, using an extended move neighbourhood with *double-cross* and *infeasible-compound* moves in addition to *relocate*, *exchange* and *cross*.

TS has to be capable of making improving and non-improving moves. The search operators for the different move-types, shown in Algorithms 6.3 to 6.5 in Section 6.1, can be used as-is without modification to return both types of moves for possible implementation. In fact, the operators have already been applied in this fashion to return non-improving, neutral moves for the extended move neighbourhood. Critical information of each move is stored in $\boldsymbol{M}$ and includes the cost of the move, ΔZ, a unique move identifier, $move_i$, and the two arcs, u and v, between which the move is applied. After the improving and non-improving moves have been returned via $\boldsymbol{M}$ the list is ordered from best to worst move. With LS, all the independent moves are then implemented in a greedy fashion using *Greedy-Compound-Moves* (Algorithm 6.9 in Section 6.3).

TS requires that move information be evaluated against information in the tabu-list, $\boldsymbol{\beta}$. Tabu moves may then not be implemented, unless certain aspiration criteria are met. In each LS iteration, *Greedy-Compound-Moves* sorts and scans the move list $\boldsymbol{M}$. The first feasible move in $\boldsymbol{M}$ is implemented, and all subsequent moves in $\boldsymbol{M}$ that are dependent on the candidate move are ignored. The process repeats with the second best feasible independent move and continues until all moves in $\boldsymbol{M}$ have been considered for implementation. A candidate move between arcs u' and v' is considered dependent on a previous move between arcs u_1 and v_1 if it disturbs any of the arcs incident to u_1 and v_1, or if it directly involves the arcs or their inverse. After an independent move between arcs u and v is made, *Update-Move-Dependence-Arc-Sets* (Algorithm 6.7) adds the two arcs as well as their incident arcs to the sets $\boldsymbol{U}_a$ and $\boldsymbol{U}_b$. A candidate move between u' and v' is dependent on an already implemented move if either of the arcs are in $\boldsymbol{U}_a$ or $\boldsymbol{U}_b$, depending on the exact move type. If it is dependent on a previous move, the move is skipped from implementation in the current LS iteration.

For TS, after a move between arcs u and v has been made, both arcs are typically added to the tabu-list $\boldsymbol{\beta}$. A candidate move between u' and v' is considered tabu if either u' or v' are in $\boldsymbol{\beta}$. This prevents the counter-move that reverses the just-implemented move. Although *Greedy-Compound-Moves* was originally developed to speed up LS, it avoids cycling as a side-effect within an LS iteration. Similar to a TS implementation, arcs that are added to $\boldsymbol{U}_a$ and $\boldsymbol{U}_b$ remain tabu until all independent, thus non-tabu moves in $\boldsymbol{M}$ have been made. *Greedy-Compound-Moves* can thus be used as-is to make improving and non-improving moves in $\boldsymbol{M}$ while enforcing tabu criteria to prevent short-term cycling. The dependent arc sets, $\boldsymbol{U}_a$ and $\boldsymbol{U}_b$, then double as tabu-lists.

If $\boldsymbol{M}$ contains a large number of non-improving moves, *all* of the independent moves will be implemented in a single LS iteration, not just the best non-improving move. This does not bode well for a TS application. Furthermore, *Greedy-Compound-Moves* does not finely control when arcs should be removed from $\boldsymbol{U}_a$. The set is simply emptied at the start of each LS iteration. Arcs involving the last move made by *Greedy-Compound-Moves* will not be made tabu, whereas arcs of the first move will be kept tabu during the entire

LS iterations. A few modifications are thus required to make *Greedy-Compound-Moves* more appropriate for a TS application.

7.2.3 Greedily compounding independent non-tabu moves

Algorithm 7.2 shows a modified version of *Greedy-Compound-Moves*, called *Tabu-Greedy-Compound-Moves*, that uses the tabu-list to prohibit tabu moves. The first modification can be found in lines 10 to 13. The tabu-list, $\boldsymbol{\beta}$, is a set containing all arcs that are tabu. If a move between u and v is to be allowed, then $u, v \notin \boldsymbol{\beta}$. If it is not allowed, the move can be flagged as either dependent on a previous move or as infeasible; either option will result in the move not being implemented. The second modification is shown in line 17. After a move between u and v is implemented, the arcs as well as their inverse arcs are added to a second tabu-list, $\boldsymbol{\beta}'$. Other moves involving either of the arcs will be disqualified in line 11 since they are dependent on the move. As a result, $\boldsymbol{\beta}$ does not have to be immediately updated.

Algorithm 7.2: *Tabu-Greedy-Compound-Moves*

```
Input  : Current solution, $\boldsymbol{T}$; savings threshold, $\Delta\overline{Z}$; savings-list, $\boldsymbol{M}$; set of tabu arcs, $\boldsymbol{\beta}$.
Output: Neighbouring solution, $\boldsymbol{T}'$, with independent moves implemented on $\boldsymbol{T}$; dependent arc set
         $\boldsymbol{U}_a$ for two-cost-link changes; dependent arc set $\boldsymbol{U}_b$ for one-cost-link changes; and the set
         of new arcs to add to the tabu-list, $\boldsymbol{\beta}'$.
1  $\boldsymbol{U}_a = \varnothing$;
2  $\boldsymbol{U}_b = \varnothing$;
3  $\Delta Z_{\text{total}} = 0$;
4  $\boldsymbol{T}' = \boldsymbol{T}$;
5  $\boldsymbol{\beta}' = \varnothing$;
6  Order $\boldsymbol{M}$ from the best to worst improving move;
7  for $\pi \in \boldsymbol{M}$ do
8      $(\Delta Z, move_i, u, v) = \pi$;
9      if $\Delta Z < \Delta\overline{Z}$ then
10         if $u \notin \boldsymbol{\beta}$ and $v \notin \boldsymbol{\beta}$ then
11             $independent =$ Check-Move-Independence$(\pi, \boldsymbol{U}_a, \boldsymbol{U}_b)$ // Algorithm 6.8 //;
12         else
13             $independent = False$;
14         if $independent = True$ then
15             $feasible =$ Check-Feasibility$(\boldsymbol{T}', u, v)$ // Algorithm 6.2 //;
16             if $feasible = True$ then
17                 $(\boldsymbol{U}_a, \boldsymbol{U}_b) =$ Update-Move-Dependence-Arc-Sets$(\pi, \boldsymbol{U}_a, \boldsymbol{U}_b)$ // Algorithm 6.7 //;
18                 Implement the move on $\boldsymbol{T}'$;
19                 $\boldsymbol{\beta}' = \boldsymbol{\beta}' \cup \{u, v\} \cup \{inv(u) : inv(u) \neq 0, inv(v) : inv(v) \neq 0\}$;
20 return $(\boldsymbol{T}', \boldsymbol{U}_a, \boldsymbol{U}_b, \boldsymbol{\beta}')$
```

After *Tabu-Greedy-Compound-Moves* has scanned and made all the feasible, independent, non-tabu moves, a second procedure aptly named *Update-Tabu-List* (shown in Algorithm 7.3) is used to update the tabu-list. Arcs of moves made in the current iteration, stored in $\boldsymbol{\beta}'$, are added to the tabu-list $\boldsymbol{\beta}$, and arcs that have been in the list for longer than the tabu-tenure, γ, are removed from $\boldsymbol{\beta}$. This requires the use of the function f_m, which returns the iteration at which arc u was last directly involved in a move. As an example, if arcs u_1 and v_1 were part of a *relocate* move in iteration $t-10$, then $f_m(u_1) = t-10$ and $f_m(v_1) = t-10$. In line 1 and 2, f_m is updated for all the arcs in $\boldsymbol{\beta}'$. In line 3 these arcs are added into the tabu-list, $\boldsymbol{\beta}_{new}$, which will be used by *Tabu-Greedy-Compound-Moves* in

Algorithm 7.3: *Update-Tabu-List*

```
Input  : Current iteration $t$; tabu tenure $\gamma$; tabu-list $\boldsymbol{\beta}$; just made moves $\boldsymbol{\beta}'$; arc move iteration
         function $f_m$.
Output: Updated tabu-list $\boldsymbol{\beta}_{new}$; and arc move iteration function $f_m$.
1 for $u \in \boldsymbol{\beta}'$ do
2     $f_m(u) = t$;
3 $\boldsymbol{\beta}_{new} = \boldsymbol{\beta}' \cup \{u \in \boldsymbol{\beta} : f_m(u) \leq t - \gamma\}$;
4 return $(\boldsymbol{\beta}_{new}, f_m)$
```

the next iteration. Also in line 3, by checking $f_m(u) \, \forall \, u \in \boldsymbol{\beta}$ against the current iteration t, all arcs that were involved in a move before iteration $t - \gamma$ are removed from the tabu-list, since they have been in the list for longer than γ iterations. The remaining arcs are added to $\boldsymbol{\beta}_{new}$ and will remain tabu in the next iteration.

7.2.4 Accelerated Tabu Search

Tabu-Greedy-Compound-Moves and *Update-Tabu-List* are the only additions needed to create the full TS algorithm shown in Algorithm 7.4. In each TS iteration, the algorithm ignores the tabu-list to see if the feasible independent improving moves result in a new incumbent solution. If so, the moves are implemented. This is done in Lines 16 to 22 by calling the normal LS *Greedy-Compound-Moves* algorithm, and by setting the move threshold to $\Delta\overline{Z} = 0$, thereby only allowing improving moves. In lines 24 to 28, if a new incumbent solution is not found, all the non-tabu feasible moves are implemented. Thereafter, in line 31, the tabu-list is updated by adding the new tabu-arcs to the list, and by removing tabu-arcs that have been in the list for more than γ iterations. The search repeats until a user-specified execution time-limit has been reached, at which point the algorithm terminates and the incumbent solution is returned.

Tabu-Greedy-Compound-Moves will make all independent non-tabu moves per iteration. It is therefore critical to establish an appropriate move cost threshold, $\Delta\overline{Z}$, that will allow TS enough freedom to explore non-improving moves, but prevent it from worsening the solution beyond repair. It should also be taken into consideration that $\Delta\overline{Z}$ limits the number of moves in $\boldsymbol{M}$, and that the computational time required to sort $\boldsymbol{M}$, and to check the feasibility, independence and tabu state of the moves increases as a function of $|\boldsymbol{M}|$,[1].

In Figure 6.6 in Section 6.4.1 we showed that a significant number LS moves have zero cost that neither improve nor worsen the solution. The neutral moves remain available at local optima and creates large, flat plateaus that have to be successfully navigated to reach new local optima. If one move is made per iteration, the tabu tenure must be set large enough to prevent TS from cycling through a small region in the plateau.

An advantage of *Tabu-Greedy-Compound-Moves* is that it can make all the independent neutral moves within a single iteration, allowing it to quickly navigate through plateaus. It also results in TS being less sensitive towards the size of the problem instance. For small instances, fewer feasible moves are available and a short tenure is required, otherwise the search becomes too restricted. With large instances, more independent moves are available per iteration, all of which will be made by *Tabu-Greedy-Compound-Moves*. As a result, *Tabu-Greedy-Compound-Moves* navigates the search far away from the current position within a single iteration. Tabu moves can then be made non-tabu after only a

[1] A slight deviation from Algorithm 6.11 that we did implement was to sort $\boldsymbol{M}$ at the start of each iteration, instead of sorting it in *Greedy-Compound-Moves*, and then needlessly again in *Tabu-Greedy-Compound-Moves*.

Algorithm 7.4: *Accelerated-Tabu-Search*

```
     Input  : Initial solution, $\boldsymbol{T}^{(0)} \in \boldsymbol{X}$; savings threshold, $\Delta\overline{Z}$; tabu tenure $\gamma$; execution time-limit,
              $\text{time}_{\text{lim}}$.
     Output: Incumbent solution, $\boldsymbol{T}^*$
 1   // tabu-search initialisation //;
 2   $t = 0$;
 3   $\boldsymbol{T}^* = \boldsymbol{T}^{(0)}$;
 4   $\boldsymbol{\beta} = \varnothing$ // initially no arcs are tabu //;
 5   for $u \in \boldsymbol{R}$ do
 6       $f_m(u) = -tau$;
 7   $\boldsymbol{M} = \varnothing$;
 8   $\boldsymbol{M}$ = Find-Relocate-Moves($\boldsymbol{T}^{(0)}, \Delta\overline{Z}, \boldsymbol{R}, \boldsymbol{R}_T, \boldsymbol{M}$) // Algorithm 6.3 //;
 9   $\boldsymbol{M}$ = Find-Exchange-Moves($\boldsymbol{T}^{(0)}, \Delta\overline{Z}, \boldsymbol{R}, \boldsymbol{R}, \boldsymbol{M}$) // Algorithm 6.4 //;
10   $\boldsymbol{M}$ = Find-Cross-Moves($\boldsymbol{T}^{(0)}, \Delta\overline{Z}, \boldsymbol{R}_T, \boldsymbol{R}_T, \boldsymbol{M}$) // Algorithm 6.5 //;
11   // $\text{time}_{\text{now}}$ is a global variable tracking the execution time of the algorithm //;
12   while $\text{time}_{\text{now}} \leq \text{time}_{\text{lim}}$ do
13       Use Reduce-Vehicles on $\boldsymbol{T}^{(t)}$ to reduce the fleet, and let $\boldsymbol{T}'$ be the result ;
14       if $|\boldsymbol{T}'| < |\boldsymbol{T}^{(t)}|$ then // the fleet size has been reduced //
15           Set $\boldsymbol{T}^{(0)} = \boldsymbol{T}'$ and return to line 1;
16       if $\boldsymbol{M} \neq \varnothing$ then
17           // first the aspiration criteria is checked //;
18           $(\boldsymbol{T}', \Delta Z_{\text{total}}, \boldsymbol{U}'_a, \boldsymbol{U}'_b)$ = Greedy-Compound-Moves($\boldsymbol{T}^{(t)}, 0, \boldsymbol{M}$) // Algorithm 6.9 //;
19           if $Z(\boldsymbol{T}') < Z(\boldsymbol{T}^*)$ or $|\boldsymbol{T}'| < |\boldsymbol{T}^*|$ then
20               $\boldsymbol{T}^* = \boldsymbol{T}'$;
21               $\boldsymbol{T}^{(t+1)} = \boldsymbol{T}'$;
22               $\boldsymbol{U}_a = \boldsymbol{U}'_a$;
23               $\boldsymbol{U}_b = \boldsymbol{U}'_b$;
24               $\boldsymbol{\beta}' = \varnothing$;
25           else
26               // non-tabu moves are made//;
27               $(\boldsymbol{T}', \boldsymbol{U}_a, \boldsymbol{U}_b, \boldsymbol{\beta}')$ = Tabu-Greedy-Compound-Moves($\boldsymbol{T}^{(t)}, \Delta\overline{Z}, \boldsymbol{M}, \boldsymbol{\beta}$) // Algorithm 7.2 //;
28               $\boldsymbol{T}^{(t+1)} = \boldsymbol{T}'$;
29               if $Z(\boldsymbol{T}') < Z(\boldsymbol{T}^*)$ or $|\boldsymbol{T}'| < |\boldsymbol{T}^*|$ then
30                   $\boldsymbol{T}^* = \boldsymbol{T}'$;
31           $(\boldsymbol{\beta}, f_m)$ = Update-Tabu-List($t, \gamma, \boldsymbol{\beta}, \boldsymbol{\beta}', f_m$) // Algorithm 7.3 //;
32           $\boldsymbol{M}'$ = Update-Savings-List($\boldsymbol{U}_a, \boldsymbol{U}_b, \boldsymbol{M}, \Delta\overline{Z}$) // Algorithm 6.10 //;
33           $\boldsymbol{M} = \boldsymbol{M}'$;
34           $t = t + 1$;
35   return ($\boldsymbol{T}^*$)
```

few iterations, as is required with small instances. Furthermore, by carefully setting $\Delta\overline{Z}$, *Tabu-Greedy-Compound-Moves* can remain close to improving solutions while navigating through plateaus.

In this thesis we chose to set $\Delta\overline{Z} = 1$, thereby allowing only neutral moves. By doing so, $\boldsymbol{M}$ is kept relatively short and parameter-tuning does not have to be performed on $\Delta\overline{Z}$. When using $\Delta\overline{Z} = 1$, the aspiration criteria becomes redundant. Any feasible move with $\Delta Z < 0$ will result in a new incumbent solution. Since $\boldsymbol{M}$ is sorted at the start of each iteration, improving candidate moves are always scanned first. The tabu criteria is only relevant to the neutral moves that follow them. By updating *Tabu-Greedy-Compound-Moves* as shown in Algorithm 7.5 the aspiration criteria can be automatically applied without having to call *Greedy-Compound-Moves*.

Algorithm 7.5: *Neutral-Tabu-Greedy-Compound-Moves*

Input : Current solution, $\boldsymbol{T}$; savings-list, $\boldsymbol{M}$; set of tabu arcs, $\boldsymbol{\beta}$.
Output: Neighbouring solution, $\boldsymbol{T}'$, with independent moves implemented on $\boldsymbol{T}$; dependent arc set $\boldsymbol{U}_a$ for two-cost-link changes; dependent arc set $\boldsymbol{U}_b$ for one-cost-link changes; and the set of new arcs to add to the tabu-list, $\boldsymbol{\beta}'$.

```
$\boldsymbol{U}_a = \varnothing$;
$\boldsymbol{U}_b = \varnothing$;
$\Delta Z_{\text{total}} = 0$;
$\boldsymbol{T}' = \boldsymbol{T}$;
$\boldsymbol{\beta}' = \varnothing$;
Order $\boldsymbol{M}$ from the best to worst improving move;
for $\pi \in \boldsymbol{M}$ do
    $(\Delta Z, move_i, u, v) = \pi$;
    if $\Delta Z < 0$ or $(u \notin \boldsymbol{\beta}$ and $v \notin \boldsymbol{\beta})$ then
        $independent =$ Check-Move-Independence$(\pi, \boldsymbol{U}_a, \boldsymbol{U}_b)$ // Algorithm 6.8 //;
        if $independent = True$ then
            $feasible =$ Check-Feasibility$(\boldsymbol{T}', u, v)$ // Algorithm 6.2 //;
            if $feasible = True$ then
                $(\boldsymbol{U}_a, \boldsymbol{U}_b) =$ Update-Move-Dependence-Arc-Sets$(\pi, \boldsymbol{U}_a, \boldsymbol{U}_b)$ // Algorithm 6.7 //;
                Implement the move on $\boldsymbol{T}'$;
                $\boldsymbol{\beta}' = \boldsymbol{\beta}' \cup \{u, v\} \cup \{inv(u) : inv(u) \neq 0, inv(v) : inv(v) \neq 0\}$;
return $(\boldsymbol{T}', \boldsymbol{U}_a, \boldsymbol{U}_b, \boldsymbol{\beta}')$
```

The final TS implementation used for our computational tests, which only makes improving and neutral moves, is shown in Algorithm 7.6. A key feature of the algorithm is that it only has two parameters[2], namely the tabu-tenure, τ, and the amount of time that it is allowed to execute, time_{lim}. It is also fully deterministic. The only other input that it requires is an initial solution $\boldsymbol{T}^{(0)}$. Minimal parametrisation is thus required before applying the algorithm.

7.3 Computational results

The aim of this chapter was to evaluate the impact of the accelerated LS heuristics within a metaheuristic application, and to determine if the metaheuristic can be used on large waste collection instances. Computational tests were performed on the *Neutral-Accelerated-Tabu-Search* (NATS) metaheuristic, linked with different constructive heuristics.

[2] The *Nearest-Neighbour-Lists* can also be activated, in which case the nearest neighbour fraction, f, will become a third parameter.

Algorithm 7.6: *Neutral-Accelerated-Tabu-Search*

```
   Input  : Initial solution, $\boldsymbol{T}^{(0)} \in \boldsymbol{X}$; tabu tenure $\gamma$; execution time-limit, $\text{time}_{\text{lim}}$.
   Output: Incumbent solution, $\boldsymbol{T}^*$
1  // tabu-search initialisation //;
2  $t = 0$;
3  $\boldsymbol{T}^* = \boldsymbol{T}^{(0)}$;
4  $\boldsymbol{\beta} = \varnothing$ // initially no arcs are tabu //;
5  for $u \in \boldsymbol{R}$ do
6      $f_m(u) = -tau$;
7  $\Delta\overline{Z} = 1$;
8  $\boldsymbol{M} = \varnothing$;
9  $\boldsymbol{M} = \textit{Find-Relocate-Moves}(\boldsymbol{T}^{(0)}, \Delta\overline{Z}, \boldsymbol{R}, \boldsymbol{R}_T, \boldsymbol{M})$ // Algorithm 6.3 //;
10 $\boldsymbol{M} = \textit{Find-Exchange-Moves}(\boldsymbol{T}^{(0)}, \Delta\overline{Z}, \boldsymbol{R}, \boldsymbol{R}, \boldsymbol{M})$ // Algorithm 6.4 //;
11 $\boldsymbol{M} = \textit{Find-Cross-Moves}(\boldsymbol{T}^{(0)}, \Delta\overline{Z}, \boldsymbol{R}_T, \boldsymbol{R}_T, \boldsymbol{M})$ // Algorithm 6.5 //;
12 // $\text{time}_{\text{now}}$ is a global variable tracking the execution time of the algorithm //;
13 while $\text{time}_{\text{now}} \leq \text{time}_{\text{lim}}$ do
14     Use Reduce-Vehicles on $\boldsymbol{T}^{(t)}$ to reduce the fleet, and let $\boldsymbol{T}'$ be the result ;
15     if $|\boldsymbol{T}'| < |\boldsymbol{T}^{(t)}|$ then // the fleet size has been reduced //
16         Set $\boldsymbol{T}^{(0)} = \boldsymbol{T}'$ and return to line 1;
17     if  $\boldsymbol{M} \neq \varnothing$ then
18         // non-tabu moves are made//;
19         $(\boldsymbol{T}', \boldsymbol{U}_a, \boldsymbol{U}_b, \boldsymbol{\beta}') = \textit{Neutral-Tabu-Greedy-Compound-Moves}(\boldsymbol{T}^{(t)}, \boldsymbol{M}, \boldsymbol{\beta})$ // Algorithm 7.5 //;
20         $\boldsymbol{T}^{(t+1)} = \boldsymbol{T}'$;
21         if $Z(\boldsymbol{T}') < Z(\boldsymbol{T}^*)$ or $|\boldsymbol{T}'| < |\boldsymbol{T}^*|$ then
22             $\boldsymbol{T}^* = \boldsymbol{T}'$;
23         $(\boldsymbol{\beta}, f_m) = \textit{Update-Tabu-List}(t, \gamma, \boldsymbol{\beta}, \boldsymbol{\beta}', f_m)$ // Algorithm 7.3 //;
24         $\boldsymbol{M}' = \textit{Update-Savings-List}(\boldsymbol{U}_a, \boldsymbol{U}_b, \boldsymbol{M}, \Delta\overline{Z})$ // Algorithm 6.10 //;
25         $\boldsymbol{M} = \boldsymbol{M}'$;
26         $t = t + 1$;
27 return $(\boldsymbol{T}^*)$
```

For the first part of our tests we relied on the *Act-IF*, *Lpr-IF* and *Cen-IF* waste collection benchmark sets. The aim of the tests was to determine if NATS is successful in improving solutions beyond the local optima of pure LS heuristics. Next we tested the performance of NATS on MCARP benchmark sets, and evaluated its solution quality against lower bound values for the instances reported in Gouveia et al. [40]. We further evaluated NATS against two other metaheuristics, namely the MCARP Memetic Algorithm of Belenguer et al. [6], and the CARPTIF Variable Neighbourhood Search algorithm of Polacek et al. [68]. For our final evaluation NATS was tested on the *Cen-IF-Full* instance with 5734 required edges and 555 required arcs, giving it a problem size of $\tau = 12\,023$. Prior to our work, the largest instance used for computational tests was *Lpr-c-05* with $\tau = 1190$.

To evaluate and compare the performance of our algorithms we used Equation (7.1) to calculate the cost gap, Z_{gap}, between an algorithm's solution $\boldsymbol{T}$ and the best known solution for the instance:

$$Z_{\text{gap}} = \frac{Z(\boldsymbol{T}) - Z_{\text{BF}}}{Z_{\text{BF}}}, \tag{7.1}$$

where Z_{BF} is the cost of the best solution found for the instance during *all* our computational tests.

We also measured the fleet size gap, K_{gap}, between $\boldsymbol{T}$ and the best known fleet size for the instance:

$$K_{\text{gap}} = |\boldsymbol{T}^*| - K_{\text{BF}}, \tag{7.2}$$

where K_{BF} is the minimum fleet size found for a problem instance during all our tests.

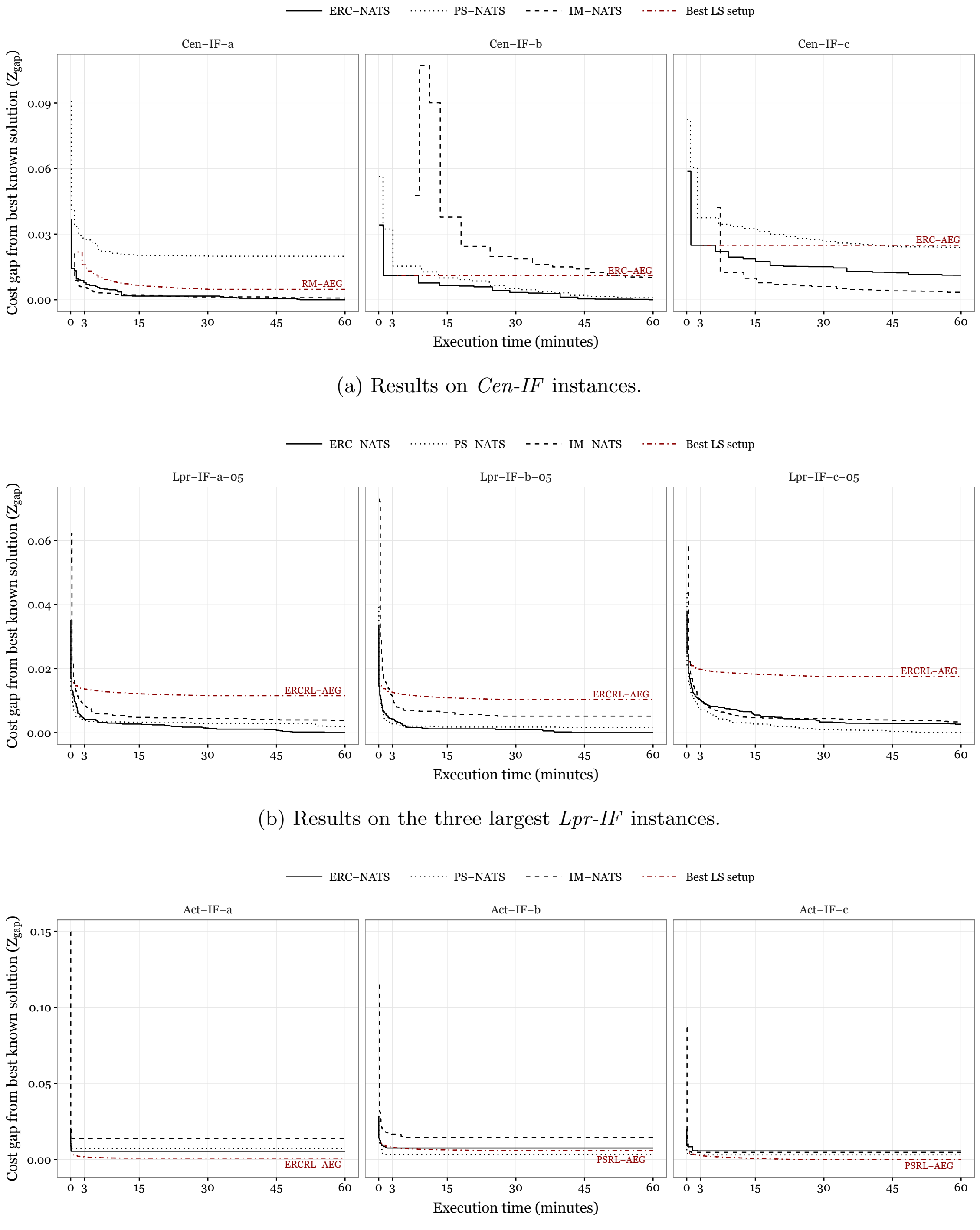
NATS takes as input two parameters, namely its execution time-limit, time_{lim} and its tabu tenure, γ. Preliminary tests showed that a tabu tenure of $\gamma = 3$ gave acceptable results on both small and large test instances. NATS was subsequently always tested with $\gamma = 3$. A description of the tests and their results can be found in Section 7.A at the end of the chapter.

All algorithms were programmed in Python version 2.7, with critical procedures optimised using Cython version 0.17.1. Computational experiments were run on a Dell PowerEdge R910 4U Rack Server with 128GB RAM with four Intel Xeon E7540 processors each having 6 cores, and 12 threads and with a 2GHz base frequency. Experiments were run without using programmatic multi-threading or multiple processors.

7.3.1 Results on the MCARPTIF instances

To evaluate the performance of NATS on the MCARPTIF waste collection tests we linked the heuristic with the three deterministic constructive heuristics, *Path-Scanning* (PS), *Improved-Merge* (IM) and *Efficient-Route-Cluster* (ERC), and set its execution time-limit to one hour, in accordance with long execution time-limits. For each of the three setups, referred to as PS-NATS, IM-NATS and ERC-NATS, we recorded the incumbent solution over its execution, allowing us to analyse the improvement of NATS over different time intervals. This allowed us to evaluate the solutions of the NATS setups at the short and medium execution time-limits, and allowed us to directly compare the setups against the best LS setups from the previous chapter.

Results for the best LS setup per problem instance, and for the NATS setups on the three largest *Lpr-IF* instances, and the *Cen-IF* and *Act-IF* sets are shown in Figure 7.1. As show in Figure 7.1a, on the large *Cen-IF* instances, NATS regularly found new incumbent solutions throughout its execution within the one hour limit. This indicates that it may



(a) Results on *Cen-IF* instances.

(b) Results on the three largest *Lpr-IF* instances.

(c) Results on the *Act-IF* instances.

Figure 7.1: Comparison between the best performing Local Search (LS) setup per problem instance and *Path-Scanning* (PS), *Improved-Merge* (IM) and *Efficient-Route-Cluster* (ERC) constructive heuristics linked with *Neutral-Accelerated-Tabu-Search* (NATS) on waste collection benchmark sets.

yet find better solutions if allowed a longer execution time. The biggest improvements between incumbent solutions were observed early on in the search. Thereafter it decreases exponentially as a function of execution time. On the smaller *Cen-IF-a* instance IM-NATS stagnated after only 15 minutes. The solution at which it stagnated was also worse than that of the RM-LS-AEG multi-start setup. Since minimising fleet size was set as the main objective, IM-NATS initially worsened the solution cost on *Cen-IF-b* to decrease the fleet size. Thereafter it was able to reduce the cost well below that of the excess fleet incumbent. With the exception of IM-NATS on *Cen-IF-a*, all the NATS setups outperformed the best LS setups, although the time required for it to do so depended on the quality of its initial solution. Over all the setups, the best initial solution resulted in the best incumbent solution. This highlights the need for developing effective constructive heuristics, even when there is sufficient time available to improve the solutions with metaheuristics.

On the *Lpr-IF* instances all the NATS setups outperformed the best LS setups, and they did so relatively quickly within 3 minutes. The biggest improvements were observed early on in the search, although NATS was still able to regularly find new incumbent solutions up-to its one hour limit. The initial solutions again influenced the quality of the final incumbent solution.

On the small *Act-IF* instances the multi-start LS setups outperformed NATS, except for PS-NATS on *Act-IF-b*, which also happens to be the largest instance in the set. On all instances NATS completely stagnated after only a few minutes. This shows the limits of NATS in that it still gets stuck in local optima on small problem instances. One option to improve its performance is to link it with multi-start constructive heuristics, thereby allowing it to improve different initial solutions under short execution time-limits.

To compare the performance of all the different heuristics developed in this thesis under short medium and long time-limits, the average Z_{gap} values of the heuristic setups were calculated over all the instances per benchmark set. The short execution time-limit was set to 3 minutes, and the medium and long time-limits were set at 30 and 60 minutes, respectively. NATS was then compared against the pure multi-start randomised constructive heuristics of Chapter 4, which we simply abbreviate as M-CH for Multi-Start-Constructive-Heuristics, as well as the deterministic and multi-start LS-Accelerated-Extended-Greedy setups of Chapter 6, respectively abbreviated as D-LS and M-LS.

The constructive heuristic setups consisted of *Efficient-Route-Cluster-Random-Link* (ERCRL), *Path-Scanning-Random-Link* (PSRL) and *Randomised-Merge* (RM). For the D-LS and NATS setups, the deterministic versions of *Efficient-Route-Cluster*, *Path-Scanning* and *Improved-Merge* were used to generate a single solution per instance, which were then improved with the setups. For the M-LS setups, the M-CH setups were used to provide different starting solutions, and each was improved to its local optimum. Since the M-CH and M-LS setups are randomised, their expected solution cost at the different execution time-limits were used to calculate Z_{gap}. The execution times of the D-LS, M-LS and NATS setups included the time to generate the initial solutions.

Summary results for the different setups on the *Cen-IF*, *Lpr-IF*, *Act-IF* and *mval-IF-3L* benchmark sets are shown in Table 7.1. Full result tables are available from Appendix A.2.1. The pure M-CH setups performed the worst, with *Randomised-Merge* being the worst among all of them. It also failed to construct a solution for the *Cen-IF-b* and *Cen-IF-c* instances within three minutes. *Efficient-Route-Cluster-Random-Link* performed the best on the *Cen-IF*, *Lpr-IF* and *Act-IF* instances, with an average Z_{gap} of between 1.6% and 4.4% on the sets. On the *mval-IF-3L* set it performed the worst. The performance of the M-CH setups did not improve by much when allowed more execution time, which shows that the additional execution time is better used on improvement heuristics.

Table 7.1: Average cost gaps, Z_{gap} in %, of different optimisation algorithms under short, medium and long execution time-limits on MCARPTIF benchmark sets.

		3 minute time-limit				30 minute time-limit				60 minute time-limit			
Set	Construct	M-CH	D-LS	M-LS	NATS	M-CH	D-LS	M-LS	NATS	M-CH	D-LS	M-LS	NATS
Cen-IF	ERC	4.4	1.7	3.3	1.5	4.1	1.7	1.8	0.7	4.1	1.7	1.8	0.4
	PS	7.1	3.1	4.3	3.3	6.6	3.1	2.6	1.7	6.6	3.1	2.6	1.5
	M	- *	- *	- *	- *	3.4	1.5	1.2	0.9	3.4	1.5	1.2	0.5
	Mean	5.1	2.1	3.1	1.8	4.7	2.1	1.9	1.1	4.7	2.1	1.9	0.8
Lpr-IF	ERC	1.8	1.3	0.8	0.3	1.8	1.3	0.6	0.2	1.8	1.3	0.6	0.2
	PS	2.5	1.3	0.8	0.4	2.5	1.3	0.7	0.3	2.5	1.3	0.7	0.3
	M	4.8	1.7	1.4	0.6	4.5	1.7	1.1	0.4	4.5	1.7	1.1	0.3
	Mean	3.0	1.4	1.0	0.4	2.9	1.4	0.8	0.3	2.9	1.4	0.8	0.3
Act-IF	ERC	1.6	1.2	0.5	0.6	1.6	1.2	0.3	0.6	1.6	1.2	0.3	0.6
	PS	1.5	0.9	0.5	0.5	1.5	0.9	0.4	0.5	1.5	0.9	0.4	0.5
	M	14.8	2.2	3.0	1.2	13.9	2.2	2.3	1.1	13.9	2.2	2.3	1.1
	Mean	6.0	1.5	1.3	0.8	5.7	1.5	1.0	0.7	5.7	1.5	1.0	0.7
mval-IF-3L	ERC	12.6	10.0	2.4	1.9	12.6	10.0	2.3	1.7	12.6	10.0	2.3	1.7
	PS	10.0	10.1	3.2	2.4	10.0	10.1	3.1	2.4	10.0	10.1	3.1	2.4
	M	10.9	11.9	3.0	3.6	10.6	11.9	2.6	3.6	10.6	11.9	2.6	3.6
	Mean	11.2	10.7	2.9	2.6	11.1	10.7	2.7	2.6	11.1	10.7	2.7	2.6
Global	ERC	5.1	3.5	1.7	1.1	5.0	3.5	1.3	0.8	5.0	3.5	1.3	0.7
	PS	5.3	3.9	2.2	1.6	5.1	3.9	1.7	1.2	5.1	3.9	1.7	1.2
	M	8.6	4.3	2.2	1.5	8.1	4.3	1.8	1.5	8.1	4.3	1.8	1.4
	Mean	6.3	3.9	2.1	1.4	6.1	3.9	1.6	1.2	6.1	3.9	1.6	1.1

M-CH: Multi-start Randomised Constructive Heuristic; D-LS: Deterministic Local Search Accelerated-Extended-Greedy; M-LS: Multi-start Local Search Accelerated-Extended-Greedy; NATS: Neutral-Accelerated Tabu Search; ERC: *Efficient-Route-Cluster*; PS: *Path-Scanning*; M: *Merge*; **Merge* failed to construct initial solutions within 3 minutes.

Under short execution times, the D-LS setups performed better than the M-LS setups on the *Cen-IF* and *Lpr-IF* sets, and yet on smaller instances they performed worse, significantly so on *mval-IF-3L* where their average Z_{gap} values were over 10%, compared to 2.9% for the M-LS setups. Since D-LS improves a single solution to its local optimum its performance remained constant over different execution time-limits. M-LS did benefit from longer execution times, and more so on the *Cen-IF* instances where its average Z_{gap} decreased from 3.1% to 1.9%.

As expected NATS performed the best out of all the setups. Of interest is that it did so even under short execution time-limits. This can be attributed to the accelerated LS mechanism from which it was developed. NATS also benefited the most from extended execution times, especially where it started with poor initial solutions. On the smaller *Act-IF* and *mval-IF-3L* sets, its performance was close to that of M-LS, with the latter producing slightly better solutions in some cases.

To evaluate the performance of the different setups in minimising the vehicle fleet size, the K_{gap} of the different setups were evaluated. We then calculated the total number of instances per benchmark set on which a setup produced a solution with an excess fleet-size, thus with $K_{\text{gap}} > 0$. Results on the *Cen-IF*, *Lpr-IF*, *Act-IF* and *mval-IF-3L* benchmark sets are shown in Table 7.2. Again, as expected, the M-CH setups performed the worst. On all 55 problem instances, *Efficient-Route-Cluster-Random-Link* produced excessive fleet-sized solutions on 23 (42%) and *Randomised-Merge* on 20 (36%) of the instances. *Path-Scanning-Random-Link* performed better, producing excess fleet-size solutions on 12 (21%) of the instances. The excess fleet-size solutions occurred despite *Reduce-Vehicles* being applied to each initial solution. NATS performed better than D-LS, but it still produced excess fleet size solutions on some of the *Lpr-IF* and *mval-IF-3L* instances. Due to their multi-start nature the M-LS setups performed the best in minimising the fleet-size over all the benchmark sets. However, under short-execution time-limits on the large *Cen-IF* instances the M-LS setups performed worse than NATS. When allowed longer execution times the setups performed the same on the benchmark set. Both NATS and M-LS performed the best when linked with *Efficient-Route-Cluster* and *Efficient-Route-Cluster-Random-Link*, respectively.

Results on the waste collection benchmark sets showed that NATS linked with *Efficient-Route-Cluster* performed the best in terms of minimising solution cost. Except for a few of the smaller *Lpr-IF* instances, the setup also performed similarly to M-LS linked with *Efficient-Route-Cluster-Random-Link* in minimising the fleet size. The role of constructive heuristics still remains important, with the quality of the initial solutions influencing the final incumbent solution at which NATS terminates. Results on the large *Cen-IF* and *Lpr-IF* instances showed that NATS is efficient enough to be used under short, medium and long execution time-limits.

7.3.2 Evaluation against existing solution approaches

A shortcoming of our evaluation of heuristics thus far is that their solution quality was internally measured against the best solutions found during all our computational tests. This is useful for identifying the best performing heuristic among those tested, but there is a risk that all the heuristics are poor, including NATS. To gain more confidence in the performance of NATS, further tests were performed on the MCARP.

The MCARPTIF can be reduced to the MCARP by setting the route duration limit sufficiently large, thereby only the vehicle capacity constraint is taken into consideration, and by setting the vehicle depot as the only available Intermediate Facility (IF). NATS can then be used as-is to solve the MCARP. It can also be used as-is on the Capacitated

Table 7.2: Number of problem instances per MCARPTIF benchmark set on which the different optimisation algorithms under short, medium and long execution time-limits produced excess-fleet solutions.

			3 minute time-limit				30 minute time-limit				60 minute time-limit			
Set	Construct	# ins	M-CH	D-LS	M-LS	DNATS	M-CH	D-LS	M-LS	DNATS	M-CH	D-LS	M-LS	DNATS
Cen-IF	ERC	3	2	0	1	0	2	0	0	0	2	0	0	0
	PS	3	2	0	2	0	1	0	0	0	1	0	0	0
	M	3	-*	-*	-*	-*	1	0	0	0	1	0	0	0
	Total	9	4	0	3	0	4	0	0	0	4	0	0	0
	Fraction		0.44	0.00	0.33	0.00	0.44	0.00	0.00	0.00	0.44	0.00	0.00	0.00
Lpr-IF	ERC	15	2	2	1	2	2	2	1	2	2	2	1	2
	PS	15	2	2	1	1	2	2	1	1	2	2	1	1
	M	15	7	2	1	1	6	2	1	1	6	2	1	1
	Total	45	11	6	3	4	10	6	3	4	10	6	3	4
	Fraction		0.24	0.13	0.07	0.09	0.22	0.13	0.07	0.09	0.22	0.13	0.07	0.09
Act-IF	ERC	3	0	0	0	0	0	0	0	0	0	0	0	0
	PS	3	0	0	0	0	0	0	0	0	0	0	0	0
	M	3	0	0	0	0	0	0	0	0	0	0	0	0
	Total	9	0	0	0	0	0	0	0	0	0	0	0	0
	Fraction		0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
mval-IF-3L	ERC	34	19	13	3	6	19	13	3	6	19	13	3	6
	PS	34	8	13	5	9	8	13	5	9	8	13	5	9
	M	34	13	20	5	13	13	20	4	13	13	20	4	13
	Total	102	40	46	13	28	40	46	12	28	40	46	12	28
	Fraction		0.39	0.45	0.13	0.27	0.39	0.45	0.12	0.27	0.39	0.45	0.12	0.27
Global	ERC	55	23	15	5	8	23	15	4	8	23	15	4	8
	PS	55	12	15	8	10	11	15	6	10	11	15	6	10
	M	55	20	22	6	14	20	22	5	14	20	22	5	14
	Total	165	55	52	19	32	54	52	15	32	54	52	15	32
	Fraction		0.33	0.32	0.12	0.19	0.33	0.32	0.09	0.19	0.33	0.32	0.09	0.19

# ins: number of problem instances per benchmark set; M-CH: Multi-start Randomised Constructive Heuristic; D-LS: Deterministic Local Search Accelerated-Extended-Greedy; M-LS: Multi-start Local Search Accelerated-Extended-Greedy; NATS: Neutral Accelerated Tabu Search; ERC: *Efficient-Route-Cluster*; PS: *Path-Scanning*; M: *Merge*; **Merge* failed to construct initial solutions within 3 minutes.

Arc Routing Problem under Time Restrictions with Intermediate Facilities (CARPTIF), thus allowing us to compare NATS against two existing metaheuristics developed for these problems, namely the MCARP Memetic Algorithm if Belenguer et al. [6] and the CARPTIF Variable Neighbourhood Search of Polacek et al. [68].

We first compare NATS against the Memetic Algorithm (MA) of Belenguer et al. [6] on the *mval* and *Lpr* MCARP benchmark sets, for which lower bound values are available from [6, 40]. Belenguer et al. [6] report on the performance of their MA for a single run under a one hour execution time-limit, and further give the costs of the best solution found during all their computational tests. We relied directly on their reported solution costs, and did not attempt to reimplement their MA. For our tests on NATS we again imposed a one hour execution time-limit, and then measured the lower bound cost gap, LB_{gap}, between the solution of NATS, $\boldsymbol{T}^*$, and the lower bound values, Z_{LB}, reported in [6, 40], using the following equation:

$$LB_{\text{gap}} = \frac{Z(\boldsymbol{T}^*) - Z_{LB}}{Z_{LB}}. \tag{7.3}$$

Results for NATS and the MA of Belenguer et al. [6] on nine *Lpr* instances are shown in Figure 7.2. The nine instances are the largest in the set. The same results with the gap measured against the previous best known solutions can be found in Section 7.B at the end of the chapter. PS-NATS required less than three minutes to outperform MA on the *Lpr-a-05*, *Lpr-b-05* and *Lpr-c-05* instances. It also found new best solutions for the instances in under five minutes. In comparison to MA, the tests confirm that NATS is extremely effective on large instances. It produced high quality solutions with gaps of less than 1.5% from lower-bound values. On the smaller *Lpr-a-04* and *Lpr-b-04* instances, the IM-NATS setup also outperformed MA, but it required more time to do so. The only instance on which it could find a new best solution was *Lpr-b-04*. It also failed to outperform MA on the *Lpr-c-04* instance within a one hour execution time-limit. All the NATS setups failed to outperform the MA on the smaller *Lpr-a-03*, *Lpr-b-03* and *Lpr-c-03* instances. The solution costs of NATS were still within 1% of the best known solutions, but similar to tests on the MCARPTIF, the algorithm seems to be less effective on small instances. Although not shown, NATS was similarly outperformed on the six smaller *Lpr-a-01* to *Lpr-c-02* instances, on which MA was able to find proven optimal solutions.

To evaluate NATS on the smaller problem instances, we linked it with *Efficient-Route-Cluster-Random-Link*, *Path-Scanning-Random-Link* and *Randomised-Merge* and allowed it a reduced execution time of between one and three minutes to improve different initial solutions. Different initial solutions were then improved until the global execution time-limit of 60 minutes was reached. We tested this setup on the *Lpr* and *mval* benchmark sets, as well as the *bccm-IF* set, used by Polacek et al. [68] to evaluate their Variable Neighbourhood Research metaheuristic. For the *Lpr* instances, the execution time per run was set to three minutes, and for the *mval* and *bccm-IF* instances it was set to one minute. Full results for the tests can be found in Appendix A.2.2.

Summary results for the deterministic and the multi-start NATS setup are shown in Table 7.3. The table shows the average LB_{gap} values of the setups over the *Lpr* and *mval* instances, and the average gap from the best solutions found by Polacek et al. [68] on the *bccm-IF* instances. On the *Lpr* and *mval* instances, the multi-start NATS setups performed better than the deterministic version. The setups further managed to find new best solutions on the smaller *Lpr-a-03* and *Lpr-b-03* instances, as well as a few *mval* instances. The multi-start NATS setups produced solutions with cost gaps of less than 1% from lower-bound values, thus indicating that on the MCARP instances NATS is capable

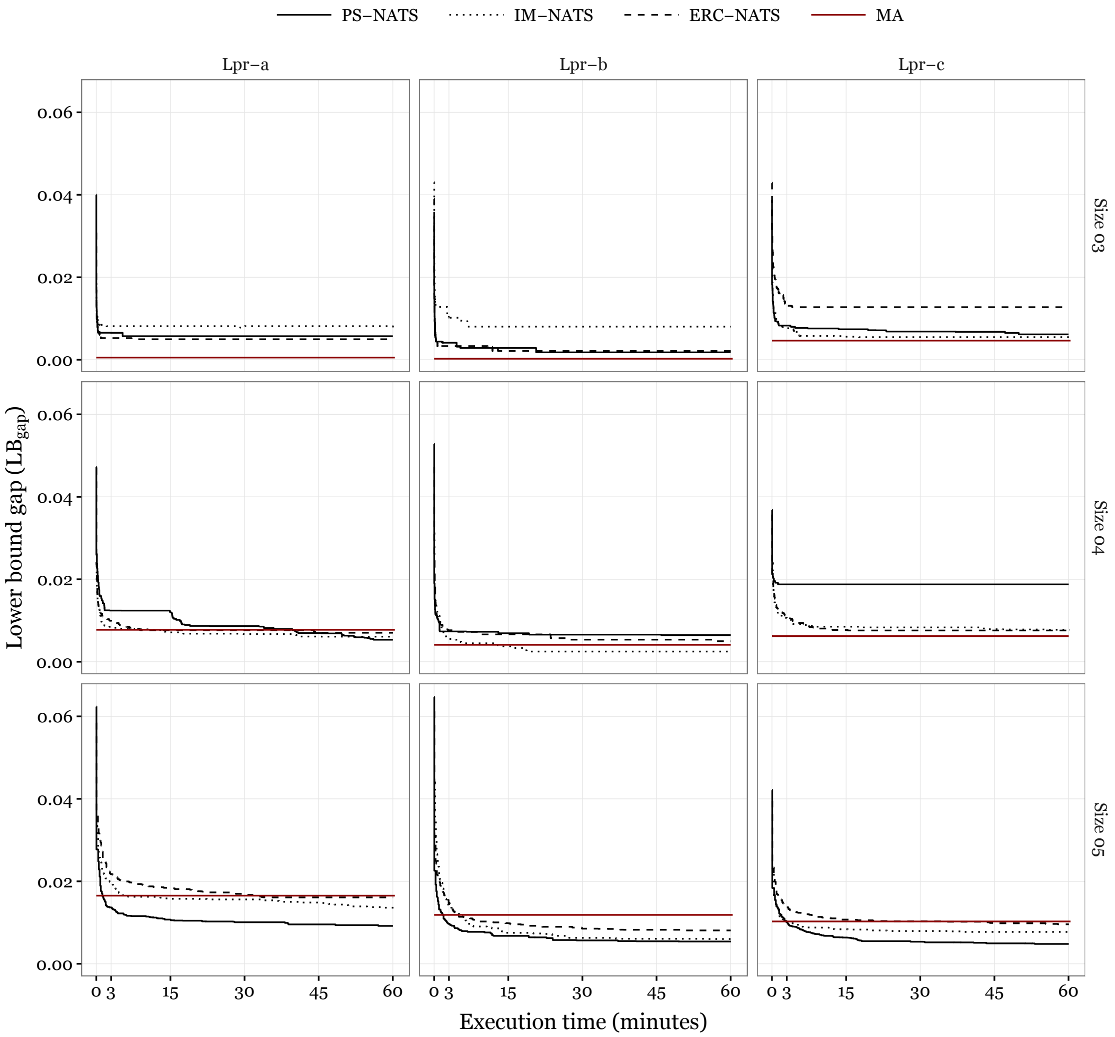


Figure 7.2: Lower-bound gaps of NATS setups, in comparison to a Memetic Algorithm [6], on nine *Lpr* instances.

Table 7.3: Average lower-bound and best-known cost gaps for NATS setups on MCARP and CARPTIF benchmark sets.

		Multi-start			Deterministic		
Set	MA [6]	PSRL-NATS	RM-NATS	ERCRL-NATS	PS-NATS	IM-NATS	ERC-NATS
Lpr	0.41	0.50	0.44	0.48	0.51	0.49	0.57
mval	0.71	0.56	0.96	0.93	2.34	2.34	3.00
*bccm-IF**		3.21	2.40	6.09	6.31	8.24	14.24
Mean		*1.42*	*1.27*	*2.50*	*3.05*	*3.69*	*5.94*

*Solutions for the Variable Neighbourhood Search heuristic reported in Polacek et al. [68] are still the best known, giving the Variable Neighbourhood Search heuristic an average best known cost gap of 0%.

of producing high quality solutions. On *bccm-IF*, the deterministic NATS setups performed extremely poorly, with average gaps of between 6% and 14% from the best known solutions. The multi-start version performed better, with an average gap of around 2.4% when linked with RM. Over all the instances, NATS linked with RM performed the best, but it is clear that NATS was not as effective on small MCARP and CARPTIF instances as the existing methods, whereas it performed extremely well on large instances.

7.3.3 Computational tests on the *Cen-IF-Full* instance

The previous tests showed that NATS can effectively deal with large *Cen-IF* instances with up-to 2755 required arcs and edges. Despite being three times larger than the *Lpr-c-05* instance, these instances are smaller than the waste collection problems often found in practice, which according to Prins [69] can contain tens of thousands of arcs.

To test the limits of our heuristics we performed a final set of tests on the *Cen-IF-Full* instance, which is a combination of all three *Cen-IF* instances. The instance contains 5734 required edges and 555 required arcs, giving it an instance size of $\tau = 12\,023$.

The algorithms were executed over four steps. First, all the pre-calculations were performed and stored for future use; second, initial solutions were generated using deterministic constructive heuristics; third, each initial solution was improved to its local optimum using LS; and lastly, each local optimum solution was further improved using NATS with a tabu tenure of $\gamma = 3$.

Before applying the heuristics we first had to pre-calculate the quickest dead-heading path times between all the arcs and edges of the instance. Thereafter the best IF to visit between all the required arcs and edges had to be calculated. The time required for this pre-calculation is seldom reported in literature, but becomes significant for huge instances. The shortest-paths for the *Cen-IF-Full* instance were calculated using our adapted *Floyd-Warshall* algorithm (Algorithm 2.1 in Section 2.A), which together with the IF calculations took 84 minutes to complete. The pre-calculation data was stored for future use, ensuring it did not have to be incurred each time a heuristic was called. This also applies in practice, assuming of course that the road network of the service area remains the same. Once stored, the pre-calculation data still had to be loaded each time computational tests were performed. This took on average 200 seconds, which already makes it impossible for the heuristics to be used within our short execution time-limit of 3 minutes.

Once the instance-data is loaded, the next step was to generate in initial solution for the instance using *Path-Scanning*, *Efficient-Route-Cluster* and *Improved-Merge*. All three heuristics were used without imposing a time limit and their solution costs compared. Results for the heuristics are shown in the top of Table 7.4. *Path-Scanning* and *Efficient-Route-Cluster* again proved to be efficient, taking 14 and 33 seconds, respectively, to generate five and four deterministic solutions, respectively, and return the best. If the loading time of the pre-calculation data can be reduced the two constructive heuristics may yet prove effective under short time-limits. The same cannot be said for the less efficient *Improved-Merge*, which took 49 minutes to generate an initial solution. It did manage to produce the lowest cost solution but the solution had a fleet size of 66 vehicles, compared to 51 vehicles required by the other two heuristic solutions

For the next step the initial solutions were improved using LS-Accelerated-Extended-Greedy (LS-ARG), again without imposing a time-limit. Results for this step are also shown in Table 7.4. On the *Path-Scanning* and *Efficient-Route-Cluster* initial solutions, LS-ARG took over 20 minutes to reach local optima, and the total savings obtained through this step was only 2% and 3%, respectively. LS-ARG took even longer on the

Table 7.4: Performance of the optimisation algorithms on the *Cen-IF-Full* instance.

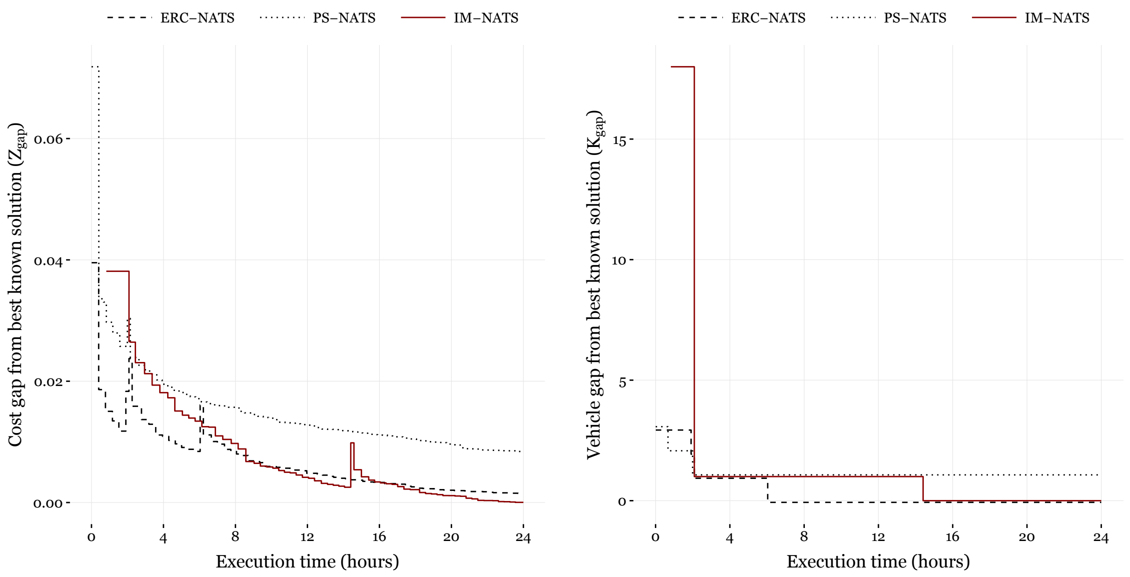
Optimisation algorithm	Performance metric	ERC	PS	IM
Constructive heuristics	Total execution time (minutes)	0.5	0.2	49
$\text{time}_{\text{lim}} = \infty$	Cost	1 405 725	1 449 403	1 403 820
	Fleet size	51	51	66
	Cost gap (Z_{gap})	4.0%	7.1%	3.8%
	Fleet gap (K_{gap})	3	3	18
Accelerated Local Search	Total execution time (minutes)	23	24	125
$\text{time}_{\text{lim}} = \infty\text{h}$	Cost	1 377 454	1 397 684	1 388 147
	Fleet size	51	51	49
	Cost gap (Z_{gap})	1.7%	3.4%	2.7%
	Fleet gap (K_{gap})	3	3	1
NATS	Total execution time (minutes)	60	60	49*
$\text{time}_{\text{lim}} = 1\text{h}$	Cost	1 372 593	1 392 452	1 403 820*
	Fleet size	51	50	66*
	Cost gap (Z_{gap})	1.5%	3.0%	7.1%*
	Fleet gap (K_{gap})	3	2	18*
NATS	Total execution time (minutes)	1440	1440	1440
$\text{time}_{\text{lim}} = 24\text{h}$	Cost	1 354 280	1 363 592	1 352 301
	Fleet size	48	49	48
	Cost gap (Z_{gap})	0.1%	0.8%	0.0%
	Fleet gap (K_{gap})	0	1	0

ERC: *Efficient-Route-Cluster*; PS: *Path-Scanning*; IM: *Improved-Merge*; NATS: Neutral Accelerated Tabu Search. *Under a 1-hour execution time-limit IM takes 49 minutes to generate a single solution, leaving too little time to complete a single TS or LS iteration, hence why the values are as given for the constructive heuristic.

Improved-Merge solution, but here it was at least capable of reducing the fleet size by 17 vehicles. In fact, after this step, the local optimum *Improved-Merge* solution required only 49 vehicles, compared to the 51 vehicles of the other solutions. The execution time of LS-ARG prevents it from being used under short time-limits, but it can be used under medium time-limits when linked with either *Path-Scanning* or *Efficient-Route-Cluster*.

The last step was to further improve the solutions using NATS. For this step we originally imposed the long execution time-limit of one hour. The time-limit includes the time to load the pre-calculation-data, to generate the initial solution and to improve it to its local optimum using LS-ARG. These three steps already took in excess of 125 minutes on the *Improved-Merge* solution. Results for NATS on the *Path-Scanning* and *Efficient-Route-Cluster* local optimum solutions are shown in Table 7.4. NATS improved the local optima solutions by a meagre 0.35% and 0.36%, although it did reduce the fleet size of the *Path-Scanning* solution from 51 to 50 vehicles. Despite the efficiency of our NATS implementation, it struggled to significantly improve the initial solutions within our medium and long execution time-limits.

To further evaluate NATS we extended its execution time to 24 hours. Results for the test are shown in Figure 7.3. Figure 7.3a shows the cost gap of NATS over its execution time and Figure 7.3b show the vehicle fleet gap. Jumps in the costs of the incumbent solutions shown in Figure 7.3a correspond with fleet reductions of Figure 7.3b, with minimising the fleet size being our primary objective. When allowed sufficient execution time, NATS was able to reduce the fleet sizes. On the *Path-Scanning* solution, this resulted in a temporary increase in solution cost, whereafter NATS was able to reduce the solution cost below its previous lowest level. The same also occurred once on the *Improved-Merge* solution. After being improved by NATS the *Improved-Merge* solution turned out to be the best. Previous tests indicated that our *Efficient-Route-Cluster* constructive heuristic is the best option for producing starting solutions. Yet, in this final test it was outper-



(a) Cost gap between the current incumbent solution of NATS and the best known solution over its execution time.

(b) Fleet size gap between the current incumbent solution of NATS and the best known solution over its execution.

Figure 7.3: Performance of NATS under a 24-hour execution time-limit on the *Cen-IF-Full* instance.

formed by *Randomised-Merge*, which happened to perform quite poorly on other waste collection instances. This shows that our NATS is still sensitive towards its initial solutions, highlighting the need for more research on constructive heuristics.

Under a 24 hour execution time-limit NATS proved to be effective on the giant *Cen-IF-Full* instance, but more research is required before the heuristic can be used for short and medium term-planning.

7.4 Conclusion

The aim of this thesis was to develop heuristics capable of generating and improving feasible solutions for the MCARPTIF under different execution time-limits. Different methods have been developed and critically evaluated in the preceding chapters, culminating in the development of our Neutral Accelerated Tabu Search metaheuristic. The metaheuristic was developed by extending the accelerated LS mechanisms from the previous chapter, and has the unique characteristic of only having two parameters, a tabu tenure and an execution time-limit. Tests on the large waste collection instances showed that it can be used under short and long execution time-limits, outperforming the pure LS setup from which it was developed. It also outperformed an existing metaheuristic on large MCARP instances. It did, however, struggle on small instances, and although its performance was improved when allowed multiple-starts, the existing metaheuristics reigned supreme on the small MCARP and CARPTIF instances.

The main aim of this chapter was determine if our research on efficient LS procedures can be easily extended to develop an efficient metaheuristic. The simplicity of our TS metaheuristic and its performance on waste collection sets showed that the extension is indeed possible, thereby creating an opportunity to improve the performance of existing LS metaheuristics on large instances. On small instances the existing heuristics are more than adequate, and difficult to improve upon. The practical relevance of further improving the performance of heuristics on small instances is questionable. We therefore recommend that more tests be performed on large instances, similar in size to those encountered in practice.

To establish the limits of our heuristics we performed further tests on a huge MCARPTIF instance. Two of our MCARPTIF constructive heuristics were capable of producing initial solutions within 30 seconds. LS then took close to 30 minutes to improve the solutions, and thereafter NATS required up-to 24 hours to significantly improve the local optimum solutions. In terms of our original goal of developing heuristics capable of generating and improving feasible solutions for the MCARPTIF under different execution time-limits, these results showed that our heuristics can be used for this purpose on realistic instances.

Chapter appendix

7.A Tabu tenure parameterisation

The following experiments were conducted to analyse the impact of the tabu tenure, γ, on the improvements made by *Neutral Accelerated Tabu Search* (NATS) on the initial solutions. NATS was used to improve three initial solutions per problem instance, generated by the *Improved-Merge*, *Path-Scanning* and *Efficient-Route-Cluster* constructive heuristics. Tests were performed on the *Act-IF*, *Cen-IF* and *Lpr-IF* waste collection benchmark sets, as well as the *mval-IF-3L* sets. NATS was tested at six tabu tenure levels: $\gamma \in \{0, 3, 5, 10, 15, 20\}$. The improvement made by NATS at each γ level to the initial solution was measured as:

$$\Delta Z^{f}_{\mathrm{TS}} = \frac{Z(\boldsymbol{T}^{(0)}) - Z(\boldsymbol{T}^{*})}{Z(\boldsymbol{T}^{(0)})}, \tag{7.4}$$

where $Z(\boldsymbol{T}^{(0)})$ is the cost of the initial solution and $Z(\boldsymbol{T}^{*})$ is the cost of the final incumbent solution returned by NATS. On each instance NATS was allowed a maximum execution time-limit of 30 minutes, and the incumbent was saved at 60 sec, 5, 15 and 30 minute intervals to determine if the best tenure changes when NATS is allowed more execution time. Results for the tests are shown in Figure 7.4 The best improvements were observed at $\gamma \in \{3, 5\}$, and the worst improvements were observed at $\gamma = 0$. This was observed over all four benchmark sets which confirms that small tabu tenures are appropriate on small and large problem instances. The difference between the different γ levels was small, indicating that NATS is not very sensitive to the tabu tenure. Based on the tests we chose to always use NATS at $\gamma = 3$, but we do note that $\gamma = 5$ could also have been used.

7.B Results for NATS on the MCARP

Figure 7.5 shows the cost gaps of NATS and the Memetic Algorithm [6] from the previously best known solutions on nine of the largest *Lpr* MCARP benchmark instances. A negative cost gap implies that NATS was able to find a new best solution for an instance. As shown in the figure, NATS achieves this very quickly on the large size 5 instances, but failed to do so on the smaller size 3 instances.

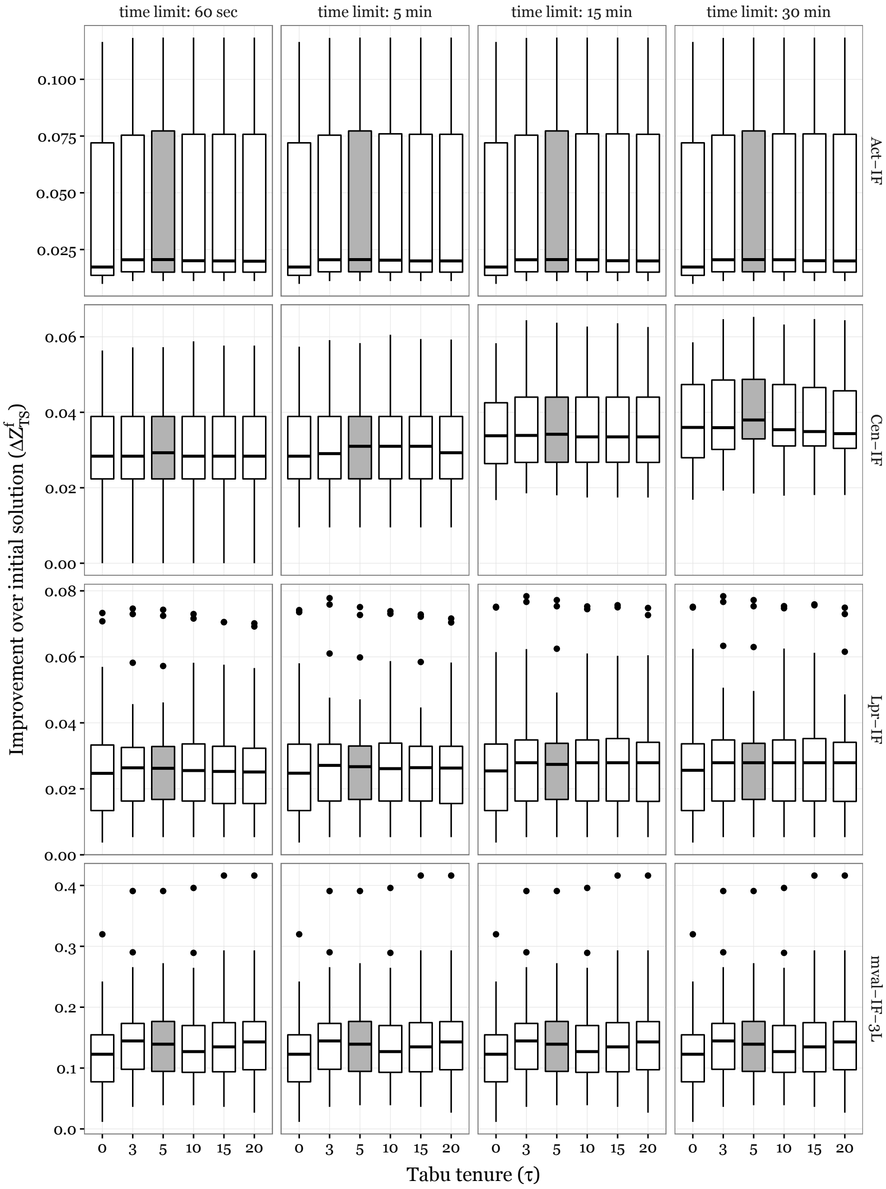


Figure 7.4: Improvement of *Neutral Accelerated Tabu Search* at different tabu tenure levels to three initial solutions per problem instance.

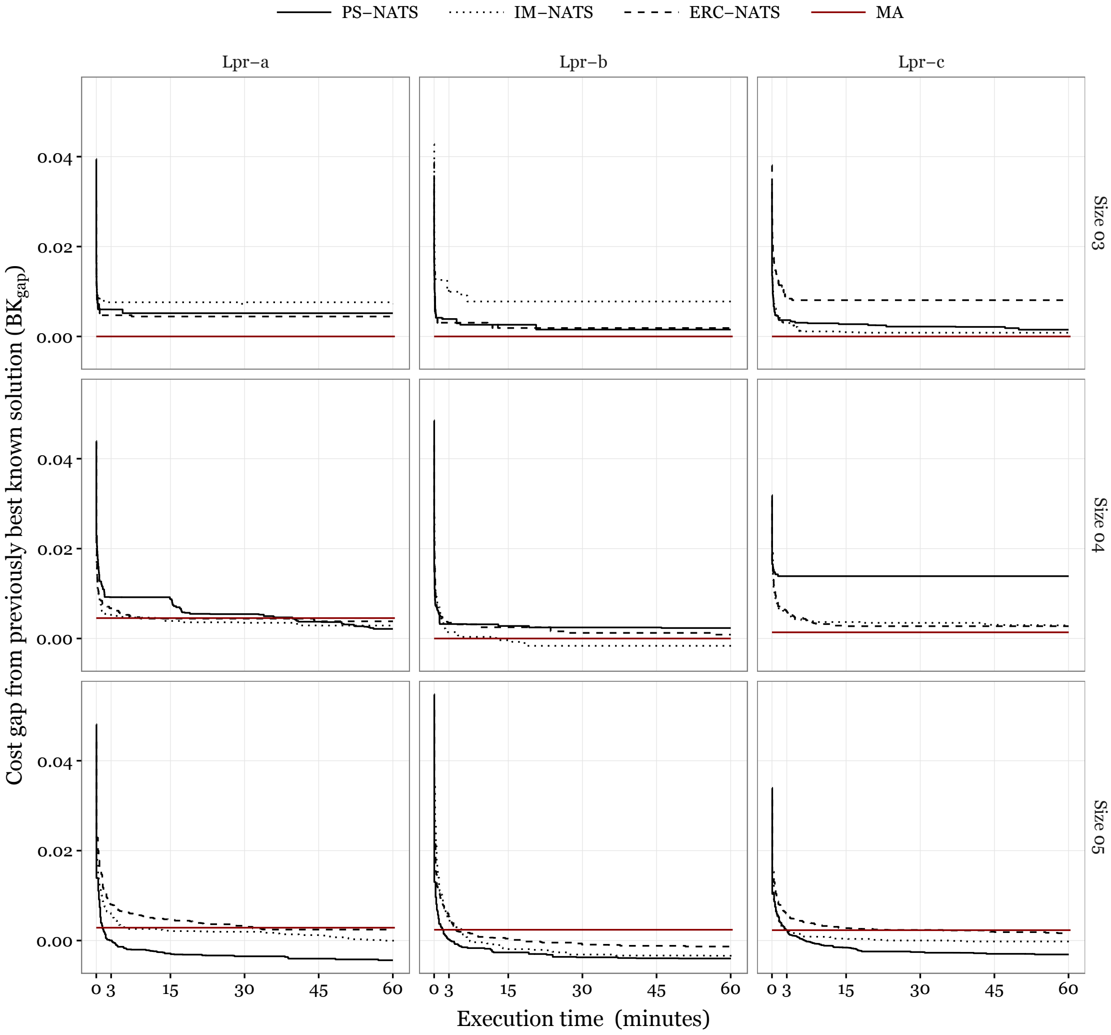


Figure 7.5: Cost gap between the NATS setups and the previous best known solutions, in comparison to the Memetic Algorithm of [6], on nine *Lpr* MCARP problem instances.

Chapter 8

Research contributions and future work

The research question that this thesis attempted to answer was: *'how should municipalities design and optimise residential waste collection routes for their waste collection vehicles?'* To answer the research question we formulated the Mixed Capacitated Arc Routing Problem under Time Restrictions with Intermediate Facilities (MCARPTIF) and formulated three specific aims for the thesis, all relating to the development of heuristics for the problem. In the next section we revisit the research aims and discuss how they were addressed in the preceding chapters. In Section 8.2 we discuss the research contributions of this thesis, focussing on the utility of our research and how the utility was demonstrated. In the last section we discus future research opportunities related to our work.

8.1 Research aims

In this thesis four aims were formulated.

(1) Find the appropriate Arc Routing Problem formulation for the MCARPTIF.

The MCARPTIF consists of elements of the Capacitated Arc Routing Problem (CARP), Mixed Capacitated Arc Routing Problem (MCARP), and Capacitated Arc Routing Problem under Time Restrictions with Intermediate Facilities (CARPTIF). The three existing problem definitions were combined into a complete formulation for the MCARPTIF problem, thus addressing the first research aim.

(2) Identify potential solution methods for the MCARPTIF under different execution time-limits.

The time-limits are applicable to situations involving short, medium and long term planning. Despite what we considered to be a common practical application, the MCARPTIF has to our knowledge not been formally studied in literature. To identify potential solution methods we therefore focussed on studies of the CARP, MCARP and CARPTIF. With all three problems being $\mathscr{NP}$-hard, and by extension so too the MCARPTIF, the most effective methods for dealing with the problems are based on heuristic and metaheuristic solution techniques. To address the second research aim we identified existing constructive heuristics that could be adapted to the MCARPTIF and used them when short execution times are available. We further identified Local Search (LS) methods for the MCARP that could be adapted and used to improve the initial solutions, and ultimately extended into a metaheuristic for the MCARPTIF.

(3) Develop heuristics capable of generating and improving feasible solutions for the MCARPTIF under different execution time-limits.

When combined, the heuristics that we developed form a three-phased approach. The contribution of each phase to the area of arc routing is as follows.

For Phase 1 we developed and tested constructive heuristics to generate initial solutions for the MCARPTIF. We first developed optimal and efficient splitting procedures and used these to develop giant tour based constructive heuristics for the problem. Next we extended two existing MCARP heuristics to deal with Intermediate Facilities (IFs). We also developed a vehicle reduction procedure that allowed the heuristics to better deal with cases where the fleet size has to be minimised. Benchmark tests were performed on realistic waste collection instances, thus improving the practical significance of our results, as well as instances developed from existing CARP, MCARP and CARPTIF benchmark sets. Results among the sets varied, with the giant-tour based heuristics performing the best on the waste collection instances. All the heuristics proved capable of quickly generating MCARPTIF solutions, thus addressing the first part of research aim three. The differences in heuristic performance among the sets highlighted the need for tests to be performed on realistic waste collection instances, few of which are currently available in literature.

For Phase 2 we developed basic LS heuristics that relied on *best-move* and *first-move* strategies. The heuristics employed five move operators, of which *relocate*, *exchange* and *cross* made the biggest contribution to the savings obtained through LS. Even when the basic LS heuristics were limited to the three main move operators, the heuristics proved to be too slow on realistically sized waste collection instances. This necessitated us to implement advanced acceleration mechanisms that enabled LS to more quickly reach local optima. The two most effective acceleration mechanisms were *Static-Move-Descriptors* and *Greedy-Compound-Independent-Moves*. Both significantly accelerated LS without negatively effecting its solution quality. Through the acceleration mechanisms, LS took at most four minutes to reach local optima on our largest test instances, whereas the basic setups took between fifteen minutes and three hours. Although LS was originally considered for situations where medium execution time-limits are available, the acceleration mechanisms made it possible for LS to be applied under short execution time-limits as well, thereby exceeding the second part of research aim three. Our work on LS is significant in the context of CARPs as all the current metaheuristics for the problems rely on some form of LS.

For Phase 3 we extended the accelerated LS heuristic to develop a Tabu Search (TS) metaheuristic, capable of further improving initial solutions beyond local optima. A key feature of the TS is that it only has two parameters, a tabu tenure and an execution time-limit. It is also fully deterministic. Tests were performed on large waste collection instances, as well as existing MCARP and CARPTIF benchmark sets, allowing the TS to be directly compared against existing solution approaches for the problems. Our main goal was to determine if our research contribution on efficient LS procedures could be easily extended to develop an efficient metaheuristic for the MCARPTIF. The simplicity of the metaheuristic and its performance on large instances showed that the extension is indeed possible, thereby creating an opportunity to improve the performance of existing LS metaheuristics on large instances. Similar to LS, we originally considered TS for situations where medium to long execution time-limits are available. The efficiency of its underlying

LS procedures made it possible for TS to be applied under short execution time-limits as well, where it outperformed the LS methods on large instances.

(4) Critically evaluate the heuristics in terms of their solution quality and execution times.

This aim was addressed throughout the development of our heuristics. To address research aim four we developed two benchmark sets, termed *Act-IF* and *Cen-IF*, that are based on actual road networks where waste collection occurs. The *Cen-IF* set contains instances with up-to 2775 required arcs and edges, whereas the largest instance prior to this thesis contained only 803. Throughout the thesis, benchmark tests were performed on the realistic waste collection instances, thus improving the practical significance of our results. The size of the *Cen-IF* instances made it possible to critically evaluate the scalability of our heuristics, and to identify inefficiencies that would have gone unnoticed if tests were limited to the smaller benchmark instances.

To fully address aim four, a last test was performed on the huge *Cen-IF-Full* waste collection instance with 6289 arcs and edges. Two of our constructive heuristics were capable of producing initial solutions within short execution time-limits. LS was then able to improve the two solutions within medium time-limits. Thereafter the TS required up-to 24 hours to significantly improve the local optimum solutions. The last test showed that the efficacy of our improvement heuristics is limited by instance size, and that more research still has to be done on the improvement heuristics to allow them to be used under short execution time-limits on huge instances.

Tests on the *Cen-IF* and *Cen-IF-Full* sets showed that the common practice of testing and proving the effectiveness of heuristics on rather small randomly generated instances does not guarantee that the heuristics will be capable of dealing with more realistic problems. By performing tests on the more realistic waste collection instances we were able to critically evaluate the heuristics in terms of their solution quality and execution times, thus addressing the fourth and final aim of this thesis.

8.2 Research contributions

The research methodology that we followed was based on the Design Research paradigm, as defined for Operations Research by Manson [56]. Design research requires the development of one or more artefacts, which in this thesis represented the heuristics that we developed for the MCARPTIF. To determine if the research contributions are significant, Manson [56] poses two questions, taken from Hevner et al. [46]. The two questions are:

> *"What utility does the new artefact(s) provide?"* and *"What demonstrates this utility?"*

Hevner et al. [46] give five testing statements through which we can evaluate our research contributions to determine their significance, and in so doing answer the above two questions.

(1) "If existing artefacts are adequate, then the production of new artefacts is unnecessary."

Existing artefacts for the MCARPTIF are not adequate for two reasons. First, to our knowledge this thesis presents the first formal study on the MCARPTIF, as such there are no existing heuristics for the problem that could have been used as-is. It was therefore necessary to adapt and extend existing heuristics for the problem, thereby creating new artefacts for the MCARPTIF. Second, the existing heuristics are not adequate when dealing with large instances. In literature, computational tests are mostly being performed on small instances, although more recently a set of medium sized instances, introduced by Brandão and Eglese [12], have also been used. Heuristics are developed to outperform existing methods on the small and medium test instances, thereby leading to competitive testing [47]. The existing heuristics have thus become extremely effective in dealing with small to medium sized instances and are becoming increasingly difficult to improve upon. However, as our tests on the MCARP instances showed, the performance of these heuristics on large instances leave room for improvement. We therefore recommend that more tests be performed on large instances, similar in size to those encountered in practice.

(2) "If the new artefact does not map adequately to the real world, it cannot provide utility."

The MCARPTIF closely models residential waste collection problems and therefore adequately maps to the real world. Despite what we considered to be a common practical application, the MCARPTIF has not been formally studied in literature. It should be noted that not all waste collection problems faced in practice deal with mixed road networks and Intermediate Facilitys (IFs). The CARP and MCARP therefore do adequately map to the real world in certain situations, but not always. An advantage of focussing on the MCARPTIF, in addition to the problem considering IFs and mixed networks, is that it can be reduced to the CARP and MCARP. The heuristics developed in this thesis can therefore be used as-is on the CARP and MCARP, allowing our artefacts to map adequately to a wider range of situations encountered in practical waste collection applications. Similar to how we extended methods for the CARP and MCARP to the MCARPTIF, our work can be extended in future studies to deal with multiple depots, heterogeneous fleets, and stochastic demand, among others.

(3) "If the artefact does not solve the problem, it has no utility."

The developed heuristics proved capable of producing high quality feasible solutions for the MCARPTIF. Finding the optimal solution for the MCARPTIF is not practically possible on large instances due to the problem being $\mathscr{NP}$-hard. For this reason we relied on heuristic solution techniques. The heuristics proved capable of generating and improving feasible solutions for the problem, which are adequate for practical applications. Another aspect that we focussed on is the vehicle fleet size. In CARP literature the fleet is often assumed to be unlimited, which is seldom the case in practice. A solution requiring more vehicles than what is available should be considered infeasible. To allow heuristics to better deal with this situation we developed a vehicle reduction heuristic that aggressively tries to reduce the vehicle fleet size, even if it results in an increase in solution cost.

(4) "If utility is not demonstrated, then there is no basis upon which to accept the claims that it provides any contribution."

In this thesis all our computational tests were performed on realistically sized waste collection instances, thereby improving and demonstrating the practical significance and research utility of our results. The heuristics were further tested under different execution time-limits, as is often encountered in practice. The ultimate goal is to demonstrate the utility of our work in the form of a case study, in which the actual improvements in the waste collection process resulting from new routes can be measured. Such a case study is dependent on heuristics capable of generating new collection routes. Through this thesis we have laid the foundations for such a case study.

The fifth and last statement is:

(5) "If the problem, the artefacts, and its utility are not presented in a manner such that the implications for research and practice are clear, then publication in the literature is not appropriate."

The final validation of our research ultimately depends on whether it is deemed acceptable for publication. We believe that our two publications in [86] and [87], based on the contents of Chapters 3 and 4, shows the significance of our research in the area of CARPs.

8.3 Future research opportunities

This thesis constitutes the first work on the MCARPTIF, which leaves much scope for future research opportunities. Specific research opportunities were referred to directly in the preceding chapters. The focus of this chapter is therefore on more general research themes.

All the existing LS based heuristics for the CARP, MCARP, and CARPTIF can be tested with our efficient LS procedures on large instances. Our developed heuristics can also be extended to deal with other realistic MCARPTIF generalisations, such as cases with heterogeneous fleets, multiple vehicle depots, and stochastic demand. The evaluation of our heuristics can also be improved, specifically by using anytime behaviour analysis as proposed by López-Ibáñez and Stützle [55].

However, we feel that a more critical and essential research contribution would be to develop more realistic benchmark problems. The *Cen-IF* and *Cen-IF-Full* sets developed in this thesis contributes towards this goal, but the instances still contain too much randomly generated information. Realistic benchmark problems are critical for developing useful heuristics capable of solving actual waste collection routing problems.

The last research opportunity that we wish to highlight is the integration of our routing heuristics into strategic optimisation models, such as determining vehicle fleet compositions, determining the location of intermediate facilities and sectoring a large collection area into collection zones. Our heuristics can then be used to provide decision support in other critical areas of waste management.

Bibliography

[1] Ahr, D. and Reinelt, G. (2015). The capacitated arc routing problem: combinatorial lower bounds. In Corberán, Á. and Laporte, G., editors, *Arc routing: problems, methods, and applications*, volume 20 of *MOS-SIAM Series on Optimization*, chapter 8, pages 159–181. SIAM.

[2] Amberg, A., Domschke, W., and Voß, S. (2000). Multiple center capacitated arc routing problems: a tabu search algorithm using capacitated trees. *European Journal of Operational Research*, 124(2):360–376.

[3] Bartolini, E., Cordeau, J.-F., and Laporte, G. (2013). Improved lower bounds and exact algorithm for the capacitated arc routing problem. *Mathematical Programming*, 137(1-2):409–452.

[4] Bautista, J., Fernández, E., and Pereira, J. (2008). Solving an urban waste collection problem using ant heuristics. *Computers & Operations Research*, 35(9):3020–3033.

[5] Beasley, J. E. (1983). Route first—cluster second methods for vehicle routing. *Omega*, 11(4):403–408.

[6] 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.

[7] Belenguer, J. M. and Benavent, E. (2003). A cutting plane algorithm for the capacitated arc routing problem. *Computers & Operations Research*, 30(5):705–728.

[8] Belenguer, J. M., Benavent, E., and Irnich, S. (2015). The capacitated arc routing problem: exact algorithms. In Corberán, Á. and Laporte, G., editors, *Arc routing: problems, methods, and applications*, volume 20 of *MOS-SIAM Series on Optimization*, chapter 9, pages 183–221. SIAM.

[9] Benavent, E., Campos, V., Corberán, A., and Mota, E. (1992). The capacitated arc routing problem: Lower bounds. *Networks*, 22(7):669–690.

[10] 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.

[11] Bode, C. and Irnich, S. (2012). Cut-first branch-and-price-second for the capacitated arc-routing problem. *Operations Research*, 60(5):1167–1182.

[12] Brandão, J. and Eglese, R. (2008). A deterministic tabu search algorithm for the capacitated arc routing problem. *Computers & Operations Research*, 35(4):1112–1126.

[13] Campbell, James F., L. A. and Perrier, N. (2015). Advances in vehicle routing for snow plowing. In Corberán, A. and Laporte, G., editors, *Arc routing: problems, methods, and applications*, volume 20 of *MOS-SIAM Series on Optimization*, chapter 15, pages 351–370. SIAM.

[14] Christiansen, C. H., Lysgaard, J., and Wøhlk, S. (2009). A branch-and-price algorithm for the capacitated arc routing problem with stochastic demands. *Operations Research Letters*, 37(6):392–398.

[15] Chu, F., Labadi, N., and Prins, C. (2006). A scatter search for the periodic capacitated arc routing problem. *European Journal of Operational Research*, 169(2):586–605.

[16] City of Cape Town (2015). Integrated annual report: 2014/15. Available online from https://www.capetown.gov.za/en/IDP/Documents [last accessed 24 April 2016].

[17] Constantino, M., Gouveia, L., Mourão, M. C., and Nunes, A. C. (2015). The mixed capacitated arc routing problem with non-overlapping routes. *European Journal of Operational Research*, 244(2):445–456.

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

[19] Corberán, A. and Prins, C. (2010). Recent results on arc routing problems: an annotated bibliography. *Networks*, 56(1):50–69.

[20] Cormen, T. H., Leiserson, C. E., and Rivest, R. L. (1990). *Introduction to algorithms*. MIT-Press.

[21] 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.

[22] De Rosa, B., Improta, G., Ghiani, G., and Musmanno, R. (2002). The arc routing and scheduling problem with transshipment. *Transportation Science*, 36(3):301–313.

[23] Del Pia, A. and Filippi, C. (2006). A variable neighborhood descent algorithm for a real waste collection problem with mobile depots. *International transactions in operational research*, 13(2):125–141.

[24] Doulabi, S. H. H. and Seifi, A. (2013). Lower and upper bounds for location-arc routing problems with vehicle capacity constraints. *European Journal of Operational Research*, 224(1):189 – 208.

[25] Dror, M., editor (2000). *Arc Routing: Theory, Solutions, and Applications*. Boston: Kluwer Academic Publishers.

[26] Eiselt, H. A., Gendreau, M., and Laporte, G. (1995a). Arc routing problems, part I: The Chinese Postman Problem. *Operations Research*, 43(2):231–242.

[27] Eiselt, H. A., Gendreau, M., and Laporte, G. (1995b). Arc routing problems, part II: The Rural Postman Problem. *Operations Research*, 43(3):399–414.

[28] 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.

[29] Gendreau, M. (2003). An introduction to tabu search. In Glower, F. and Kochenberger, G., editors, *Handbook of Metaheuristics*. Kluwer Academic Publishers.

[30] Ghiani, G., Guerriero, F., Improta, G., and Musmanno, R. (2005). Waste collection in Southern Italy: solution of a real-life arc routing problem. *International Transactions in Operational Research*, 12(2):135–144.

[31] 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.

[32] Ghiani, G., Improta, G., and Laporte, G. (2001). The capacitated arc routing problem with intermediate facilities. *Networks*, 37(3):134–143.

[33] Ghiani, G., Laganá, D., Laporte, G., and Mari, F. (2010). Ant colony optimization for the arc routing problem with intermediate facilities under capacity and length restrictions. *Journal of Heuristics*, 16(2):211–233.

[34] Ghiani, G. and Laporte, G. (2015). The undirected rural postman problem. In Corberán, Á. and Laporte, G., editors, *Arc routing: problems, methods, and applications*, volume 20 of *MOS-SIAM Series on Optimization*, chapter 5, pages 85–99. SIAM.

[35] Ghiani, G., Mourão, M. C., Pinto, L., and Vigo, D. (2015). Routing in waste collection applications. In Corberán, Á. and Laporte, G., editors, *Arc routing: problems, methods, and applications*, volume 20 of *MOS-SIAM Series on Optimization*, chapter 15, pages 351–370. SIAM.

[36] Glover, F. (1986). Future paths for integer programming and links to artificial intelligence. *Computers & Operations Research*, 13(5):533–549.

[37] Glower, F. and Kochenberger, G., editors (2003). *Handbook of Metaheuristic*. Kluwer Academic Publishers.

[38] Golden, B. L., DeArmon, J. S., and Baker, E. K. (1983). Computational experiments with algorithms for a class of routing problems. *Computers & Operations Research*, 10(1):47–59.

[39] Golden, B. L. and Wong, R. T. (1981). Capacitated arc routing problems. *Networks*, 11(3):305–315.

[40] 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.

[41] Grandinetti, L., Guerriero, F., Laganà, D., and Pisacane, O. (2012). An optimization-based heuristic for the multi-objective undirected capacitated arc routing problem. *Computers & Operations Research*, 39(10):2300–2309.

[42] Greistorfer, P. (2003). A tabu scatter search metaheuristic for the arc routing problem. *Computers & Industrial Engineering*, 44(2):249–266.

[43] Groves, G., Le Roux, J., and Van Vuuren, J. (2004). Network service scheduling and routing. *International transaction in operational research*, 11(6):613–643.

[44] Hertz, A., Laporte, G., and Mittaz, M. (2000). A tabu search heuristic for the capacitated arc routing problem. *Operations Research*, 48(1):129–135.

[45] Hertz, A. and Mittaz, M. (2001). A variable neighborhood descent algorithm for the undirected capacitated arc routing problem. *Transportation Science*, 35(4):425–435.

[46] Hevner, A. R., March, S. T., Park, J., and Ram, S. (2004). Design science in information systems research. *MIS quarterly*, 28(1):75–105.

[47] Hooker, J. N. (1995). Testing heuristics: We have it all wrong. *Journal of Heuristics*, 1(1):33–42.

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

[49] Karadimas, N. V., Papatzelou, K., and Loumos, V. G. (2007). Optimal solid waste collection routes identified by the ant colony system algorithm. *Waste Management & Research*, 25(6):139–147.

[50] Krushinsky, D. and Van Woensel, T. (2015). An approach to the asymmetric multi-depot capacitated arc routing problem. *European Journal of Operational Research*, 244(1):100–109.

[51] Lacomme, P., Prins, C., and Ramdane-Chérif, W. (2004). Competitive memetic algorithms for arc routing problems. *Annals of Operations Research*, 131(4):159–185.

[52] Laporte, G. (2015). The undirected chinese postam problem. In Corberán, Á. and Laporte, G., editors, *Arc routing: problems, methods, and applications*, volume 20 of *MOS-SIAM Series on Optimization*, chapter 3, pages 53–64. SIAM.

[53] Li, L. Y. and Eglese, R. W. (1996). An interactive algorithm for vehicle routeing for winter-gritting. *Journal of the Operational Research Society*, 47(2):217–228.

[54] Liu, T., Jiang, Z., and Geng, N. (2013). A memetic algorithm with iterated local search for the capacitated arc routing problem. *International Journal of Production Research*, 51(10):3075–3084.

[55] López-Ibáñez, M. and Stützle, T. (2014). Automatically improving the anytime behaviour of optimisation algorithms. *European Journal of Operational Research*, 235(3):569–582.

[56] Manson, N. (2006). Is operations research really research? *ORiON: The Journal of ORSSA*, 22(2):155–180.

[57] Martinelli, R., Poggi, M., and Subramanian, A. (2013). Improved bounds for large scale capacitated arc routing problem. *Computers & Operations Research*, 40(8):2145–2160.

[58] Martinez, C., Loiseau, I., Resende, M., and Rodriguez, S. (2011). BRKGA algorithm for the capacitated arc routing problem. *Electronic Notes in Theoretical Computer Science*, 281:69–83.

[59] Mei, Y., Li, X., and Yao, X. (2014). Cooperative co-evolution with route distance grouping for large-scale capacitated arc routing problems. *IEEE Transactions on Evolutionary Computation*, 18(3):435–449.

[60] Mourão, M. C. and Almeida, M. T. (2000). Lower-bounding and heuristic methods for a refuse collection vehicle routing problem. *European Journal of Operational Research*, 121(2):420–434.

[61] Mourão, M. C. and Amado, L. (2005). Heuristic method for a mixed capacitated arc routing problem: a refuse collection application. *European Journal of Operational Research*, 160(1):139–153.

[62] Mourão, M. C., Nunes, A. C., and Prins, C. (2009). Heuristic methods for the sectoring arc routing problem. *European Journal of Operational Research*, 196(3):856–868.

[63] Muyldermans, L. and Pang, G. (2010). A guided local search procedure for the multi-compartment capacitated arc routing problem. *Computers & Operations Research*, 37(9):1662–1673.

[64] 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.

[65] National Treasury (2011). *Local Government Budgets and Expenditure Review: 2006/07–2012/13*. Republic of South Africa. Available online from `http://www.treasury.gov.za/publications/igfr/2011/lg/default.aspx` [last accessed 24 April 2016].

[66] Nemhauser, G., Kan, A. R., and Todd, M. (1989). Preface. In G.L. Nemhauser, A. R. K. and Todd, M., editors, *Optimization*, volume 1 of *Handbooks in Operations Research and Management Science*, pages v – ix. Elsevier.

[67] Pearn, W. L. (1989). Approximate solutions for the capacitated arc routing problem. *Computers & Operations Research*, 16(6):589–600.

[68] Polacek, M., Doerner, Karl F. Hartl, R. F., and Maniezzo, V. (2008). A variable neighborhood search for the capacitated arc routing problem with intermediate facilities. *Journal of Heuristics*, 14(5):405–423.

[69] 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.

[70] Prins, C., Lacomme, P., and Prodhon, C. (2014). Order-first split-second methods for vehicle routing problems: A review. *Transportation Research Part C: Emerging Technologies*, 40:179–200.

[71] Rardin, R. (1997). *Optimization in Operations Research*. Prentice Hall.

[72] Rardin, R. L. and Uzsoy, R. (2001). Experimental evaluation of heuristic optimization algorithms: a tutorial. *Journal of Heuristics*, 7(3):261–304.

[73] Rodrigues, A. M. and Soeiro Ferreira, J. (2015). Waste collection routing—limited multiple landfills and heterogeneous fleet. *Networks*, 65(2):155–165.

[74] Sahoo, S., Kim, S., Kim, B.-I., Kraas, B., and Popov Jr., A. (2005). Routing optimization for waste management. *Interfaces*, 35(1):24–36.

[75] Santos, L., Coutinho-Rodrigues, J., and Antunes, C. H. (2011). A web spatial decision support system for vehicle routing using google maps. *Decision Support Systems*, 51(1):1–9.

[76] Santos, L., Coutinho-Rodrigues, J., and Current, J. R. (2008). Implementing a multi-vehicle multi-route spatial decision support system for efficient trash collection in Portugal. *Transportation Research Part A: Policy and Practice*, 42(6):922–934.

[77] Santos, L., Coutinho-Rodrigues, J., and Current, J. R. (2009). An improved heuristic for the capacitated arc routing problem. *Computers & Operations Research*, 36(9):2632–2637.

[78] 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.

[79] Sörensen, K. (2015). Metaheuristics—the metaphor exposed. *International Transactions in Operational Research*, 22(1):3–18.

[80] Talbi, E. (2009). *Metaheuristics: From design to implementation*. Wiley, New Jersey.

[81] Tang, K., Mei, Y., and Yao, X. (2009). Memetic algorithm with extended neighborhood search for capacitated arc routing problems. *IEEE Transactions on Evolutionary Computation*, 13:1151–1166.

[82] Ulusoy, G. (1985). The fleet size and mix problem for capacitated arc routing. *European Journal of Operational Research*, 22(3):329–337.

[83] Usberti, F. L., França, P. M., and França, A. L. M. (2013). Grasp with evolutionary path-relinking for the capacitated arc routing problem. *Computers & Operations Research*, 40(12):3206–3217.

[84] Viotti, P., Pelettini, A., Pomi, R., and Innocetti, C. (2003). Genetic algorithms as a promising tool for optimisation of the MSW collection routes. *Waste Management Research*, 21(4):292–298.

[85] Willemse, E. J. and Joubert, J. W. (2012). Applying min $-$ max k postmen problems to the routing of security guards. *Journal of the Operational Research Society*, 63(2):245–260.

[86] 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.

[87] 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.

[88] 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*, DOI: `10.17632/9x4vd92rcj.3`. Available online from `http://dx.doi.org/10.17632/9x4vd92rcj.3`.

[89] 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.

[90] Winston, W. and Venkataramanan, M. (2003). *Introduction to mathematical programming*, volume one of *Operations research*. California: Duxbury.

[91] Wøhlk, S. (2008). A decade of capacitated arc routing. In Golden, B., Raghavan, S., and Wasil, E., editors, *The Vehicle Routing Problem: Latest Advances and New Challenges*, volume 43 of *Operations Research/Computer Science Interfaces Series*, pages 29–48. Springer US.

[92] 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.

Appendices

Appendix A

Detailed results tables

A.1 Full results for constructive heuristics

This section contains full result tables, referred to in Chapter 4, for the MCARPTIF constructive heuristics. Heuristics tested include Path-Scanning (PS), Path-Scanning-Random-Link (PSRL), Improved-Merge (IM), Randomised-Merge (RM), Efficient-Route-Cluster (ERC), Efficient-Route-Cluster-Random-Link (ERCRL), Route-Cluster (RC) and Route-Cluster-Random-Link (RCRL).

A.1.1 Results on *Act-IF*, *Cen-IF* and *lpr-IF*

Results for all heuristics for *Act-IF*, *Cen-IF* and *Lpr-IF* problem instances are shown in Table A.1. Results for deterministic heuristic versions are shown for cost, Z, and number of required vehicles, K. For the randomised versions results are shown for the expected values of $\overline{Z}$ and $\overline{K}$, when the heuristics are allowed 100 runs ($\alpha = 100$). Computational times of the heuristics for the sets are shown in Table A.2.

Table A.1: Heuristic results for *Act-IF*, *Cen-IF* and *Lpr-IF* sets

		Best found		IM		RM(100)		PS		PSRL(100)		ERC		ERCRL(100)		RC		RCRL(100)	
1st Obj	Instance	Z	K	Z	K	$\overline{Z}$	$\overline{K}$	Z	K	$\overline{Z}$	$\overline{K}$	Z	K	$\overline{Z}$	$\overline{K}$	Z	K	$\overline{Z}$	$\overline{K}$
min Z	Cen-IF-a	231431	9	235650	9	233135	10	251726	9	250258	9	237433	13	239082	16	237433	11	239005	12
	Cen-IF-b	594142	21	597825	30	596913	31	610737	22	610799	22	597253	26	601669	28	594142	27	599486	26
	Cen-IF-c	512669	19	515952	20	514160	23	550367	20	548107	20	526067	31	529738	33	526067	22	529574	25
	Mean Z_{gap} and K_{gap}			1.0 %	3.3	0.5 %	5	6.3 %	0.7	5.9 %	0.7	1.9 %	7	2.6 %	9.3	1.7 %	3.7	2.5 %	4.7
	Act-IF-a	22351	1	25398	1	26457	1	22499	1	22470	1	22553	1	22587	1	22553	1	22533	1
	Act-IF-b	72244	3	87808	4	87054	4	73040	3	72816	3	73151	3	72789	3	72821	3	72598	3
	Act-IF-c	49972	2	54911	3	55415	2	50400	2	50209	2	50445	2	50189	2	50296	2	50125	2
	Mean Z_{gap} and K_{gap}			15.0 %	0.7	16.6 %	0.3	0.9 %	0	0.6 %	0	1.0 %	0	0.8 %	0	0.8 %	0	0.5 %	0
	Lpr-IF-a-01	13594	1	13954	1	13877	1	13783	1	13717	1	13810	2	13718	1	13713	1	13639	1
	Lpr-IF-a-02	28399	1	29113	2	28786	1	29516	2	28789	1	28966	2	28718	1	28720	1	28551	1
	Lpr-IF-a-03	77670	3	78565	4	78095	4	80261	3	79775	3	78896	4	78840	5	78834	4	78712	4
	Lpr-IF-a-04	132824	5	142214	7	152995	7	133006	5	133746	5	133840	6	133432	6	133733	6	133166	6
	Lpr-IF-a-05	211345	8	216907	10	227939	10	212252	8	212548	8	212295	9	212213	9	211840	9	211784	8
	Lpr-IF-b-01	14835	1	15261	1	15122	1	15009	1	14915	1	14927	1	14914	1	14927	1	14875	1
	Lpr-IF-b-02	28773	1	29196	2	29078	2	29955	2	29752	2	29230	2	29085	2	29230	2	29031	2
	Lpr-IF-b-03	79664	3	80215	3	80268	4	80786	3	80897	3	80690	5	80124	5	80624	4	79998	4
	Lpr-IF-b-04	131493	5	136204	6	139558	6	134580	5	133794	5	132940	6	132689	6	132387	6	132267	6
	Lpr-IF-b-05	219670	8	224653	10	234169	10	222874	8	222684	8	221413	9	220783	10	220782	9	220305	9
	Lpr-IF-c-01	18735	1	19004	1	18910	1	18837	1	18811	1	18845	1	18814	1	18807	1	18771	1
	Lpr-IF-c-02	36605	2	37338	2	37317	2	36903	2	36834	2	36858	2	36797	2	36737	2	36700	2
	Lpr-IF-c-03	113648	4	117256	5	118103	6	114908	4	114450	4	114293	5	114330	5	114127	5	114067	5
	Lpr-IF-c-04	172944	7	174065	8	173674	8	176151	7	176470	7	174438	9	174010	10	174255	9	173783	9
	Lpr-IF-c-05	271558	10	277458	11	279447	13	276587	10	276887	10	272676	13	272257	14	272271	13	271981	13
	Mean Z_{gap} and K_{gap}			2.4 %	0.9	3.6 %	1.1	1.7 %	0.1	1.4 %	0.1	1.0 %	1.1	0.7 %	1.2	0.8 %	0.9	0.4 %	0.8
min K	Cen-IF-a	231431	9	235650	9	233476	9	251726	9	250258	9	242929	9	242680	9	238868	9	240410	9
	Cen-IF-b	594142	21	605708	29	609014	25	610737	22	610799	22	599029	22	605166	22	596721	22	601452	21
	Cen-IF-c	512669	19	529887	19	515652	19	550367	20	547883	20	528713	20	539099	19	532141	19	534676	19

Mean Z_{gap} and K_{gap}			2.4 %	2.7	1.3 %	1.3	6.3 %	0.7	5.9 %	0.7	3.0 %	0.7	4.0 %	0.3	2.5 %	0.3	3.1 %	0
Act-IF-a	22351	1	25398	1	26457	1	22499	1	22470	1	22553	1	22587	1	22553	1	22533	1
Act-IF-b	72244	3	79179	3	85858	3	73040	3	72816	3	73151	3	72789	3	72821	3	72598	3
Act-IF-c	49972	2	53771	2	54916	2	50400	2	50209	2	50445	2	50189	2	50296	2	50125	2
Mean Z_{gap} and K_{gap}			10.3 %	0	15.7 %	0	0.9 %	0	0.6 %	0	1.0 %	0	0.8 %	0	0.8 %	0	0.5 %	0
Lpr-IF-a-01	13594	1	13954	1	13877	1	13783	1	13717	1	13816	1	13718	1	13713	1	13639	1
Lpr-IF-a-02	28399	1	28678	1	28726	1	28776	1	28789	1	28966	2	28692	1	28720	1	28551	1
Lpr-IF-a-03	77670	3	79775	3	78521	3	80261	3	79775	3	79576	3	79382	3	79103	3	78958	3
Lpr-IF-a-04	132824	5	140752	6	150247	6	133006	5	133746	5	134452	5	133838	5	133838	5	133234	5
Lpr-IF-a-05	211345	8	216390	9	226016	9	212252	8	212548	8	213366	8	212801	8	212300	8	211784	8
Lpr-IF-b-01	14835	1	15261	1	15122	1	15009	1	14915	1	14927	1	14914	1	14927	1	14875	1
Lpr-IF-b-02	28773	1	29196	2	29078	2	29955	2	29752	2	29230	2	29085	2	29230	2	29031	2
Lpr-IF-b-03	79664	3	80215	3	80992	3	80786	3	80897	3	81338	3	80818	3	80872	3	80131	3
Lpr-IF-b-04	131493	5	140297	5	141052	5	134580	5	133794	5	133763	5	133226	5	132840	5	132363	5
Lpr-IF-b-05	219670	8	230112	9	231697	9	222874	8	222684	8	222582	8	222835	8	220837	8	220728	8
Lpr-IF-c-01	18735	1	19004	1	18910	1	18837	1	18811	1	18845	1	18814	1	18807	1	18771	1
Lpr-IF-c-02	36605	2	37338	2	37317	2	36903	2	36834	2	36858	2	36797	2	36737	2	36700	2
Lpr-IF-c-03	113648	4	117256	5	117345	5	114763	4	114450	4	114293	5	114846	4	114715	4	114067	4
Lpr-IF-c-04	172944	7	177241	7	174195	7	176151	7	176470	7	175302	7	174765	7	174590	7	174034	7
Lpr-IF-c-05	271558	10	280380	10	281448	11	276587	10	276887	10	276318	10	275637	10	274146	10	273463	10
Mean Z_{gap} and K_{gap}			2.9 %	0.3	3.5 %	0.4	1.5 %	0.1	1.4 %	0.1	1.4 %	0.2	1.1 %	0.1	0.9 %	0.1	0.5 %	0.1

Table A.2: Computational times, in seconds, of heuristics to complete 100 runs ($\alpha = 100$) for *Cen-IF* and *Lpr-IF*, *Act-IF* sets.

Instance	α	IM	RM	PS	PSRL	ERC	ERCRL	RC	RCRL
Cen-IF-a	100	48.2	4818.3	0.8	9.5	1.1	18.0	15.6	259.5
Cen-IF-b	100	478.3	47830.6	5.4	67.3	7.1	117.7	68.3	1138.5
Cen-IF-c	100	393.6	39364.4	7.1	89.0	9.6	160.2	63.0	1050.0
Mean time		306.7	30671.1	4.4	55.3	5.9	98.6	49.0	816.0
Act-IF-a	1000	1.0	98.8	0.0	0.5	0.1	1.2	1.4	23.0
Act-IF-b	1000	8.5	851.1	0.2	1.9	0.3	5.3	8.5	141.3
Act-IF-c	1000	2.9	293.4	0.1	1.8	0.2	2.5	3.2	53.5
Mean time		4.1	414.4	0.1	1.4	0.2	3.0	4.4	72.6
Lpr-IF-a-01	100	0.1	5.3	0.0	0.3	0.0	0.4	0.1	1.4
Lpr-IF-a-02	100	0.2	23.6	0.1	1.8	0.1	1.2	0.4	7.1
Lpr-IF-a-03	100	2.3	225.4	0.1	1.2	0.2	3.4	2.6	43.3
Lpr-IF-a-04	100	5.9	585.5	0.2	2.3	0.3	5.4	4.8	80.7
Lpr-IF-a-05	100	9.7	971.5	0.4	4.5	0.7	11.3	8.5	141.9
Lpr-IF-b-01	100	0.0	2.9	0.0	0.1	0.0	0.3	0.1	1.2
Lpr-IF-b-02	100	0.1	13.3	0.1	0.9	0.0	0.8	0.4	6.5
Lpr-IF-b-03	100	1.4	143.4	0.1	1.2	0.2	3.5	2.7	44.3
Lpr-IF-b-04	100	3.2	319.6	0.2	2.6	0.4	6.1	5.0	83.1
Lpr-IF-b-05	100	10.2	1018.8	0.3	4.2	0.6	10.2	8.2	135.9
Lpr-IF-c-01	100	0.1	8.2	0.0	0.2	0.0	0.3	0.1	1.1
Lpr-IF-c-02	100	0.3	33.4	0.0	0.4	0.0	0.8	0.3	5.1
Lpr-IF-c-03	100	3.6	361.6	0.3	4.2	0.3	4.9	1.6	26.5
Lpr-IF-c-04	100	9.2	917.2	0.2	3.1	0.4	6.7	3.1	51.7
Lpr-IF-c-05	100	19.6	1961.7	0.5	5.8	0.7	12.2	5.5	91.7
Mean time		4.4	439.4	0.2	2.2	0.3	4.5	2.9	48.1

A.1.2 Results on small benchmark sets

Heuristics tested in Chapter 4 were *Improved-Merge* (IM), *Randomised-Merge* (RM), *Path-Scanning* (PS), *Path-Scanning-Random-Link* (PSRL), *Efficient-Route-Cluster* (ERC), *Efficient-Route-Cluster-Random-Link* (ERCRL), *Route-Cluster* (RC) and *Route-Cluster-Random-Link* (RCRL). Results for all heuristics for *mval-IF-3L*, *bccm-IF-3L*, *gdb-IF-3L*, *bccm-IF-3L* and *gdb-IF* problem instances are shown in Tables A.3 to A.7. Results for deterministic heuristic versions are shown for cost, Z, and number of required vehicles, K. For the randomised versions results are shown for the expected values of $\overline{Z}$ and $\overline{K}$, when the heuristics are allowed 1000 runs ($\alpha = 1000$). Computational times of the heuristics for all the sets are shown in Tables A.8 to A.12.

Table A.3: Heuristic results for *mval-IF-3L* set

		Best found		IM		RM(1000)		PS		PSRL(1000)		ERC		ERCRL(1000)		RC		RCRL(1000)	
1^{st} Obj	Instance	Z	K	Z	K	$\overline{Z}$	$\overline{K}$	Z	K	$\overline{Z}$	$\overline{K}$	Z	K	$\overline{Z}$	$\overline{K}$	Z	K	$\overline{Z}$	$\overline{K}$
min Z	mval-IF-3L-1A	250	2	312	3	294	3	281	3	250	2	276	3	268	3	276	3	268	3
	mval-IF-3L-1B	309	3	360	4	316	3	349	3	336	3	362	4	336	3	356	3	317	3
	mval-IF-3L-1C	356	3	452	6	387	4	448	4	398	4	426	5	397	4	410	5	377	4
	mval-IF-3L-2A	413	2	545	3	525	3	511	3	442	2	523	3	494	3	523	3	493	3
	mval-IF-3L-2B	429	2	488	3	477	3	517	3	442	2	490	3	483	3	490	3	481	3
	mval-IF-3L-2C	506	3	691	5	582	3	551	3	524	3	619	3	555	3	586	3	528	3
	mval-IF-3L-3A	136	2	166	3	158	3	148	2	139	2	149	2	138	2	149	2	138	2
	mval-IF-3L-3B	149	2	173	3	173	3	179	3	151	2	174	3	154	2	172	3	154	2
	mval-IF-3L-3C	123	2	188	4	147	2	149	2	131	2	150	2	140	2	146	2	131	2
	mval-IF-3L-4A	645	3	749	4	717	4	766	4	698	3	751	4	722	4	741	4	714	4
	mval-IF-3L-4B	757	4	829	4	804	4	825	4	791	4	791	4	777	4	791	4	759	4
	mval-IF-3L-4C	769	4	864	6	867	5	834	4	802	4	856	5	798	4	846	4	788	4
	mval-IF-3L-4D	782	4	886	5	820	4	902	4	832	4	906	5	839	4	865	4	790	4
	mval-IF-3L-5A	790	4	991	6	990	5	813	4	806	4	853	4	819	4	853	4	819	4
	mval-IF-3L-5B	743	4	935	5	963	5	799	4	762	4	815	4	776	4	815	4	773	4
	mval-IF-3L-5C	824	4	976	6	1054	6	885	4	841	4	952	5	937	5	950	5	935	5
	mval-IF-3L-5D	821	4	1020	6	1004	6	890	4	845	4	920	5	898	5	902	5	878	4
	mval-IF-3L-6A	347	3	384	3	357	3	410	3	387	3	397	3	378	3	397	3	369	3
	mval-IF-3L-6B	334	3	389	4	345	3	432	3	393	3	394	4	358	4	376	3	358	3
	mval-IF-3L-6C	419	3	483	4	427	4	549	4	503	4	549	6	482	6	525	4	473	4
	mval-IF-3L-7A	390	4	423	4	394	4	454	4	427	4	438	5	421	4	438	5	421	4
	mval-IF-3L-7B	447	4	518	5	471	5	543	5	512	5	536	7	494	5	536	6	488	5
	mval-IF-3L-7C	494	4	546	5	505	5	614	5	573	5	571	8	536	7	560	6	526	6
	mval-IF-3L-8A	668	4	714	4	687	4	694	4	686	4	698	4	675	4	698	4	675	4
	mval-IF-3L-8B	591	3	692	4	630	4	607	3	610	3	669	4	644	4	663	4	632	3
	mval-IF-3L-8C	660	4	744	6	670	5	732	4	696	4	769	6	735	4	734	5	697	4
	mval-IF-3L-9A	543	4	693	7	654	6	620	5	591	5	655	6	603	5	655	6	603	5
	mval-IF-3L-9B	527	4	639	6	637	6	597	5	574	5	600	5	588	5	600	5	587	5
	mval-IF-3L-9C	528	4	605	5	633	6	585	5	547	4	598	5	571	5	598	5	570	5
	mval-IF-3L-9D	649	5	711	6	696	6	715	6	681	5	738	6	719	7	714	6	686	6
	mval-IF-3L-10A	833	5	896	6	959	7	871	5	850	5	926	6	868	5	926	6	868	5
	mval-IF-3L-10B	820	5	951	7	935	6	855	5	829	5	907	6	849	5	907	6	849	5
	mval-IF-3L-10C	782	5	796	5	871	6	843	5	806	5	832	5	801	5	825	5	794	5

	mval-IF-3L-10D	795	5	892	7	881	6	841	5	809	5	884	7	846	6	863	6	813	5
	Mean Z_{gap} and K_{gap}			18.0 %	1.3	12.4 %	0.9	13.6 %	0.4	6.2 %	0.2	14.7 %	1.1	8.3 %	0.6	13.0 %	0.7	6.4 %	0.4
min K	mval-IF-3L-1A	250	2	312	3	277	3	281	3	250	2	276	3	268	3	276	3	268	3
	mval-IF-3L-1B	309	3	344	3	315	3	349	3	332	3	366	3	334	3	356	3	317	3
	mval-IF-3L-1C	356	3	434	5	380	4	448	4	398	4	435	4	397	4	413	4	377	3
	mval-IF-3L-2A	413	2	545	3	501	2	433	2	430	2	523	3	494	2	523	3	493	2
	mval-IF-3L-2B	429	2	488	3	475	2	517	3	436	2	490	3	483	3	490	3	481	2
	mval-IF-3L-2C	506	3	632	4	543	3	551	3	524	3	619	3	555	3	586	3	528	3
	mval-IF-3L-3A	136	2	161	2	151	2	144	2	139	2	149	2	138	2	149	2	138	2
	mval-IF-3L-3B	149	2	173	3	171	2	179	3	151	2	174	3	154	2	172	3	154	2
	mval-IF-3L-3C	123	2	205	3	134	2	149	2	131	2	150	2	140	2	146	2	131	2
	mval-IF-3L-4A	645	3	697	3	707	3	718	3	684	3	751	4	722	3	741	4	714	3
	mval-IF-3L-4B	757	4	829	4	780	4	825	4	791	4	791	4	777	4	791	4	759	4
	mval-IF-3L-4C	769	4	884	5	822	4	834	4	802	4	868	4	798	4	846	4	788	4
	mval-IF-3L-4D	782	4	922	4	806	4	898	4	832	4	906	5	839	4	865	4	790	4
	mval-IF-3L-5A	790	4	896	5	948	4	813	4	806	4	853	4	819	4	853	4	819	4
	mval-IF-3L-5B	743	4	825	4	925	4	799	4	762	4	815	4	776	4	815	4	773	4
	mval-IF-3L-5C	824	4	996	5	1017	5	875	4	838	4	952	5	937	4	950	5	935	4
	mval-IF-3L-5D	821	4	987	5	952	5	890	4	845	4	920	5	898	4	902	5	878	4
	mval-IF-3L-6A	347	3	384	3	357	3	410	3	387	3	397	3	378	3	397	3	369	3
	mval-IF-3L-6B	334	3	368	3	345	3	432	3	391	3	400	3	358	3	376	3	358	3
	mval-IF-3L-6C	419	3	483	4	431	3	549	4	503	4	553	4	488	4	525	4	473	4
	mval-IF-3L-7A	390	4	423	4	394	4	454	4	427	4	440	4	421	4	440	4	421	4
	mval-IF-3L-7B	447	4	518	5	458	4	543	5	503	4	557	5	494	4	545	5	488	4
	mval-IF-3L-7C	494	4	546	5	505	5	610	5	573	5	575	6	536	5	619	5	532	5
	mval-IF-3L-8A	668	4	714	4	679	4	694	4	686	4	698	4	675	4	698	4	675	4
	mval-IF-3L-8B	591	3	692	4	618	3	607	3	606	3	669	4	644	3	663	4	632	3
	mval-IF-3L-8C	660	4	741	5	670	4	732	4	696	4	785	4	735	4	742	4	697	4
	mval-IF-3L-9A	543	4	660	5	634	5	620	5	591	5	656	5	603	4	656	5	603	4
	mval-IF-3L-9B	527	4	597	5	622	5	597	5	543	4	600	5	588	5	600	5	587	5
	mval-IF-3L-9C	528	4	605	5	609	5	585	5	539	4	598	5	571	5	598	5	570	4
	mval-IF-3L-9D	649	5	711	6	674	5	715	6	677	5	738	6	717	6	714	6	686	5
	mval-IF-3L-10A	833	5	893	5	913	5	871	5	850	5	926	6	868	5	926	6	868	5
	mval-IF-3L-10B	820	5	902	6	907	5	855	5	829	5	907	6	849	5	907	6	849	5
	mval-IF-3L-10C	782	5	796	5	838	5	843	5	806	5	832	5	801	5	825	5	794	5
	mval-IF-3L-10D	795	5	967	6	851	5	841	5	809	5	889	6	846	5	863	6	813	5

	Mean Z_{gap} and K_{gap}			16.3 %	0.7	9.1 %	0.2	12.6 %	0.3	5.7 %	0.1	15.2 %	0.6	8.3 %	0.2	13.5 %	0.6	6.4 %	0.1

Table A.4: Heuristic results for *bccm-IF-3L* set

		Best found		IM		RM(1000)		PS		PSRL(1000)		ERC		ERCRL(1000)		RC		RCRL(1000)	
1^{st} Obj	Instance	Z	K	Z	K	$\overline{Z}$	$\overline{K}$	Z	K	$\overline{Z}$	$\overline{K}$	Z	K	$\overline{Z}$	$\overline{K}$	Z	K	$\overline{Z}$	$\overline{K}$
min Z	bccm-IF-3L-1A	189	2	258	3	230	2	201	2	190	2	191	2	190	2	191	2	191	2
	bccm-IF-3L-1B	189	2	259	4	225	2	208	2	196	2	207	2	196	2	199	2	190	2
	bccm-IF-3L-1C	237	2	310	7	251	3	260	3	244	2	285	4	255	3	271	3	248	3
	bccm-IF-3L-2A	264	2	345	2	299	2	293	2	278	2	297	2	282	2	288	2	282	2
	bccm-IF-3L-2B	274	2	361	3	281	2	304	2	286	2	304	2	290	2	288	2	280	2
	bccm-IF-3L-2C	357	2	498	5	375	2	400	2	363	2	430	3	398	2	412	2	379	2
	bccm-IF-3L-3A	92	2	111	2	106	2	97	2	93	2	102	2	93	2	102	2	93	2
	bccm-IF-3L-3B	93	2	113	2	106	2	105	2	94	2	104	2	94	2	102	2	93	2
	bccm-IF-3L-3C	103	2	156	6	110	2	118	2	107	2	124	2	109	2	118	2	106	2
	bccm-IF-3L-4A	418	2	547	4	519	3	474	2	439	2	486	3	438	2	472	2	435	2
	bccm-IF-3L-4B	419	2	553	5	510	3	475	2	445	2	492	3	454	2	492	3	446	2
	bccm-IF-3L-4C	426	2	565	6	516	3	458	2	451	2	526	3	455	2	492	3	440	2
	bccm-IF-3L-4D	505	3	615	7	552	3	577	3	533	3	586	4	536	3	542	3	507	3
	bccm-IF-3L-5A	494	3	744	4	667	4	541	3	509	3	560	3	520	3	542	3	517	3
	bccm-IF-3L-5B	496	3	746	5	656	4	548	3	509	3	558	3	527	3	548	3	517	3
	bccm-IF-3L-5C	510	3	762	6	668	4	560	3	521	3	596	3	530	3	568	3	531	3
	bccm-IF-3L-5D	568	3	790	7	700	4	584	3	579	3	648	3	620	3	638	3	591	3
	bccm-IF-3L-6A	233	2	287	3	245	2	260	2	247	2	266	3	246	2	258	2	244	2
	bccm-IF-3L-6B	249	2	285	2	260	2	286	2	272	2	291	3	263	2	283	3	258	2
	bccm-IF-3L-6C	359	3	401	4	359	3	419	3	397	3	421	4	399	4	411	4	390	4
	bccm-IF-3L-7A	295	3	363	4	318	3	348	3	329	3	332	3	313	3	326	3	310	3
	bccm-IF-3L-7B	309	3	371	4	319	3	355	3	333	3	326	3	322	3	326	3	315	3
	bccm-IF-3L-7C	378	4	438	5	379	4	424	4	412	4	439	6	406	6	428	5	395	4
	bccm-IF-3L-8A	411	2	563	4	514	3	472	3	422	2	460	3	418	2	460	3	419	2
	bccm-IF-3L-8B	410	2	563	5	523	3	445	3	422	2	476	3	452	3	460	3	438	3
	bccm-IF-3L-8C	495	3	614	8	539	3	557	3	516	3	560	4	522	3	529	3	514	3
	bccm-IF-3L-9A	366	3	513	5	481	4	387	3	375	3	419	4	390	3	402	3	390	3
	bccm-IF-3L-9B	378	3	513	5	481	4	397	3	381	3	426	4	406	3	424	4	394	3
	bccm-IF-3L-9C	380	3	527	7	481	5	400	3	390	3	446	4	419	4	430	4	413	3

	bccm-IF-3L-9D	460	4	571	7	524	5	492	4	466	4	518	5	490	4	504	5	478	4
	bccm-IF-3L-10A	476	3	648	5	627	5	511	3	490	3	535	4	499	3	522	3	499	3
	bccm-IF-3L-10B	487	3	648	5	639	4	505	3	497	3	538	4	519	3	538	4	497	3
	bccm-IF-3L-10C	490	3	656	5	631	4	527	3	503	3	565	4	536	3	556	4	519	3
	bccm-IF-3L-10D	586	4	689	7	661	4	606	4	591	4	641	4	616	4	635	4	605	4
	Mean Z_{gap} and K_{gap}			31.6 %	2.2	17.3 %	0.6	10.1 %	0.1	4.0 %	0.0	14.0 %	0.6	6.1 %	0.2	10.8 %	0.4	4 %	0.1
min K	bccm-IF-3L-1A	189	2	220	2	230	2	201	2	190	2	191	2	190	2	191	2	191	2
	bccm-IF-3L-1B	189	2	264	3	225	2	208	2	196	2	207	2	196	2	199	2	190	2
	bccm-IF-3L-1C	237	2	309	6	251	2	260	3	244	2	298	3	255	3	271	3	247	2
	bccm-IF-3L-2A	264	2	345	2	299	2	293	2	278	2	297	2	282	2	288	2	282	2
	bccm-IF-3L-2B	274	2	372	2	281	2	304	2	286	2	304	2	290	2	288	2	280	2
	bccm-IF-3L-2C	357	2	452	4	375	2	400	2	363	2	430	3	398	2	412	2	379	2
	bccm-IF-3L-3A	92	2	111	2	106	2	97	2	93	2	102	2	93	2	102	2	93	2
	bccm-IF-3L-3B	93	2	113	2	106	2	105	2	94	2	104	2	94	2	102	2	93	2
	bccm-IF-3L-3C	103	2	131	5	110	2	118	2	107	2	124	2	109	2	118	2	106	2
	bccm-IF-3L-4A	418	2	529	3	519	2	474	2	439	2	486	3	438	2	472	2	435	2
	bccm-IF-3L-4B	419	2	542	4	510	2	475	2	445	2	492	3	454	2	492	3	446	2
	bccm-IF-3L-4C	426	2	676	4	516	2	458	2	451	2	526	3	455	2	492	3	440	2
	bccm-IF-3L-4D	505	3	614	6	552	3	577	3	533	3	590	3	536	3	542	3	507	3
	bccm-IF-3L-5A	494	3	744	4	667	3	541	3	509	3	560	3	520	3	542	3	517	3
	bccm-IF-3L-5B	496	3	691	4	656	3	548	3	509	3	558	3	527	3	548	3	517	3
	bccm-IF-3L-5C	510	3	722	4	668	3	560	3	521	3	596	3	530	3	568	3	531	3
	bccm-IF-3L-5D	568	3	818	5	700	3	584	3	579	3	648	3	620	3	638	3	591	3
	bccm-IF-3L-6A	233	2	299	2	245	2	260	2	247	2	272	2	246	2	258	2	244	2
	bccm-IF-3L-6B	249	2	285	2	260	2	286	2	272	2	299	2	263	2	287	2	258	2
	bccm-IF-3L-6C	359	3	437	3	359	3	419	3	397	3	423	3	399	3	417	3	388	3
	bccm-IF-3L-7A	295	3	337	3	318	3	348	3	329	3	332	3	313	3	326	3	310	3
	bccm-IF-3L-7B	309	3	371	4	319	3	355	3	333	3	326	3	322	3	326	3	315	3
	bccm-IF-3L-7C	378	4	490	4	379	4	424	4	412	4	484	4	410	4	454	4	395	4
	bccm-IF-3L-8A	411	2	540	3	514	3	472	3	419	2	460	3	418	2	460	3	419	2
	bccm-IF-3L-8B	410	2	529	4	523	3	421	2	421	2	476	3	452	2	460	3	438	2
	bccm-IF-3L-8C	495	3	624	7	539	3	557	3	516	3	580	3	522	3	529	3	514	3
	bccm-IF-3L-9A	366	3	439	4	481	4	387	3	375	3	419	4	390	3	402	3	390	3
	bccm-IF-3L-9B	378	3	461	4	481	4	397	3	381	3	426	4	406	3	424	4	394	3
	bccm-IF-3L-9C	380	3	485	5	481	4	399	3	390	3	446	4	419	3	430	4	413	3
	bccm-IF-3L-9D	460	4	565	6	524	4	492	4	466	4	518	5	490	4	520	4	478	4

	bccm-IF-3L-10A	476	3	567	4	627	4	511	3	490	3	535	4	499	3	522	3	499	3
	bccm-IF-3L-10B	487	3	628	4	639	4	505	3	497	3	538	4	519	3	538	4	497	3
	bccm-IF-3L-10C	490	3	653	4	631	4	527	3	503	3	565	4	536	3	556	4	519	3
	bccm-IF-3L-10D	586	4	678	5	661	4	606	4	591	4	641	4	613	4	635	4	605	4
	Mean Z_{gap} and K_{gap}			28.4 %	1.2	17.3 %	0.2	9.9 %	0.1	4.0 %	0.0	14.8 %	0.4	6.2 %	0.0	11.2 %	0.3	4.0 %	0.0

Table A.5: Heuristic results for *gdb-IF-3L* set

		Best found		IM		RM(1000)		PS		PSRL(1000)		ERC		ERCRL(1000)		RC		RCRL(1000)	
1^{st} Obj	Instance	Z	K	Z	K	$\overline{Z}$	$\overline{K}$	Z	K	$\overline{Z}$	$\overline{K}$	Z	K	$\overline{Z}$	$\overline{K}$	Z	K	$\overline{Z}$	$\overline{K}$
min Z	gdb-IF-3L-1	308	2	316	3	308	2	337	2	310	2	338	4	318	2	337	3	308	2
	gdb-IF-3L-2	337	2	369	5	345	2	362	2	341	2	361	3	347	2	353	3	338	2
	gdb-IF-3L-3	275	2	301	4	276	2	281	2	283	2	297	3	284	2	289	2	276	2
	gdb-IF-3L-4	344	2	427	4	383	2	377	2	349	2	424	2	369	2	396	2	369	2
	gdb-IF-3L-5	448	2	520	4	468	2	490	2	458	2	528	3	495	2	516	3	466	2
	gdb-IF-3L-6	293	2	341	4	301	2	313	2	293	2	313	2	303	2	313	2	294	2
	gdb-IF-3L-7	342	2	349	4	343	2	349	2	343	2	387	3	351	2	387	3	342	2
	gdb-IF-3L-8	419	3	449	5	422	4	471	4	448	4	470	4	458	4	466	4	446	4
	gdb-IF-3L-9	343	3	374	6	361	3	388	3	365	3	402	4	372	3	381	4	352	3
	gdb-IF-3L-10	312	2	374	3	323	2	355	2	317	2	336	2	322	2	336	2	316	2
	gdb-IF-3L-11	447	3	532	4	477	4	495	3	465	3	509	4	481	3	505	3	469	3
	gdb-IF-3L-12	624	2	826	3	710	3	662	2	641	2	777	3	682	3	727	3	659	2
	gdb-IF-3L-13	594	2	634	3	600	2	640	2	609	2	640	3	604	2	623	2	602	2
	gdb-IF-3L-14	110	2	134	6	112	3	132	3	114	2	132	3	120	2	125	3	112	2
	gdb-IF-3L-15	58	1	62	3	58	1	62	2	60	1	62	2	58	1	60	1	58	1
	gdb-IF-3L-16	131	4	139	5	131	4	141	4	133	4	139	4	133	4	139	4	133	4
	gdb-IF-3L-17	93	3	97	5	93	3	99	3	95	3	97	3	95	3	97	3	95	3
	gdb-IF-3L-18	178	3	210	6	194	4	186	3	180	3	197	3	188	3	190	3	188	3
	gdb-IF-3L-19	83	1	91	1	83	1	91	1	91	1	93	1	83	1	93	1	83	1
	gdb-IF-3L-20	147	2	162	3	150	2	147	2	147	2	165	3	155	2	160	3	155	2
	gdb-IF-3L-21	164	4	171	6	166	4	170	4	166	4	174	5	166	5	172	5	166	4
	gdb-IF-3L-22	215	5	229	8	219	6	226	5	218	5	227	6	220	6	225	6	220	6
	gdb-IF-3L-23	243	6	258	9	245	7	255	6	250	6	259	8	251	8	255	7	249	7
	Mean Z_{gap} and K_{gap}			11.7 %	1.9	3.0 %	0.3	7.8 %	0.1	2.6 %	0.0	11.3 %	0.8	4.5 %	0.3	8.8 %	0.5	2.5 %	0.1

min K	gdb-IF-3L-1	308	2	366	2	308	2	337	2	310	2	358	2	318	2	338	2	308	2
	gdb-IF-3L-2	337	2	371	4	345	2	362	2	341	2	361	3	347	2	365	2	338	2
	gdb-IF-3L-3	275	2	336	3	276	2	281	2	283	2	345	2	284	2	289	2	276	2
	gdb-IF-3L-4	344	2	422	3	383	2	377	2	349	2	424	2	369	2	396	2	369	2
	gdb-IF-3L-5	448	2	511	3	464	2	490	2	458	2	528	3	495	2	516	3	466	2
	gdb-IF-3L-6	293	2	375	3	301	2	313	2	293	2	313	2	303	2	313	2	294	2
	gdb-IF-3L-7	342	2	386	3	343	2	349	2	343	2	387	3	351	2	387	3	342	2
	gdb-IF-3L-8	419	3	446	4	422	3	471	4	450	3	470	4	458	3	466	4	446	3
	gdb-IF-3L-9	343	3	386	3	358	3	388	3	365	3	402	4	372	3	402	3	352	3
	gdb-IF-3L-10	312	2	366	2	312	2	355	2	317	2	336	2	322	2	336	2	316	2
	gdb-IF-3L-11	447	3	495	3	465	3	495	3	465	3	519	3	481	3	505	3	469	3
	gdb-IF-3L-12	624	2	769	3	681	2	662	2	641	2	777	3	682	2	727	3	659	2
	gdb-IF-3L-13	594	2	653	2	600	2	640	2	609	2	644	2	604	2	623	2	602	2
	gdb-IF-3L-14	110	2	129	4	112	2	132	3	112	2	132	3	120	2	125	3	112	2
	gdb-IF-3L-15	58	1	64	2	58	1	62	2	60	1	62	2	58	1	60	1	58	1
	gdb-IF-3L-16	131	4	141	4	131	4	141	4	133	4	139	4	133	4	139	4	133	4
	gdb-IF-3L-17	93	3	103	4	93	3	99	3	95	3	97	3	95	3	97	3	95	3
	gdb-IF-3L-18	178	3	196	4	188	3	186	3	180	3	197	3	188	3	190	3	188	3
	gdb-IF-3L-19	83	1	91	1	83	1	91	1	91	1	93	1	83	1	93	1	83	1
	gdb-IF-3L-20	147	2	160	2	148	2	147	2	147	2	165	3	151	2	165	2	151	2
	gdb-IF-3L-21	164	4	176	4	166	4	170	4	166	4	178	4	166	4	174	4	166	4
	gdb-IF-3L-22	215	5	228	7	219	5	226	5	218	5	227	6	219	5	229	5	219	5
	gdb-IF-3L-23	243	6	267	7	247	6	255	6	250	6	261	7	253	6	255	7	249	6
	Mean Z_{gap} and K_{gap}			13.3 %	0.7	2.3 %	0.0	7.8 %	0.1	2.5 %	0.0	12.6 %	0.5	4.4 %	0.0	9.6 %	0.3	2.3 %	0.0

Table A.6: Heuristic results for *bccm-IF* set

		Best found		IM		RM(1000)		PS		PSRL(1000)		ERC		ERCRL(1000)		RC		RCRL(1000)	
1^{st} Obj	Instance	Z [1]	K	Z	K	$\overline{Z}$	$\overline{K}$	Z	K	$\overline{Z}$	$\overline{K}$	Z	K	$\overline{Z}$	$\overline{K}$	Z	K	$\overline{Z}$	$\overline{K}$
min Z	bccm-IF-1A	229	6	301	9	261	7	309	8	240	6	365	10	309	8	365	10	309	8
	bccm-IF-1B	229	6	301	9	261	7	309	8	242	6	365	10	309	8	365	10	309	8
	bccm-IF-1C	259	8	326	10	287	8	347	9	303	8	390	11	337	9	390	11	330	9

[1] Best solution costs include those reported in [68].

	bccm-IF-2A	434	6	480	7	434	6	507	7	441	6	743	11	687	10	743	11	687	10
	bccm-IF-2B	434	6	480	7	436	6	507	7	441	6	743	11	681	10	743	11	681	10
	bccm-IF-2C	488	7	595	9	533	8	612	9	558	8	863	13	754	11	863	13	754	11
	bccm-IF-3A	120	5	138	6	129	5	145	6	128	5	174	7	153	6	174	7	153	6
	bccm-IF-3B	120	5	138	6	128	5	134	5	128	5	174	7	157	6	174	7	157	6
	bccm-IF-3C	126	5	164	8	141	6	168	7	143	6	201	8	173	7	199	8	164	7
	bccm-IF-4A	533	8	632	9	617	9	775	10	625	8	939	14	767	10	939	14	767	10
	bccm-IF-4B	533	8	632	9	614	9	770	10	630	8	939	14	812	11	939	14	812	11
	bccm-IF-4C	533	8	632	9	619	9	766	10	625	8	939	14	790	11	939	14	790	11
	bccm-IF-4D	538	9	692	10	640	10	697	9	689	9	974	14	839	12	974	14	842	12
	bccm-IF-5A	879	13	1103	16	990	14	1083	15	1007	14	1848	27	1608	23	1848	27	1608	23
	bccm-IF-5B	879	13	1103	16	991	14	1083	15	1011	14	1848	27	1599	23	1848	27	1599	23
	bccm-IF-5C	879	13	1103	16	994	14	1083	15	1006	14	1848	27	1596	23	1848	27	1596	23
	bccm-IF-5D	879	13	1103	16	996	14	1085	15	1009	14	1848	27	1616	23	1848	27	1616	23
	bccm-IF-6A	343	7	377	8	348	7	410	8	357	7	448	9	387	8	448	9	387	8
	bccm-IF-6B	343	7	377	8	349	7	410	8	361	7	448	9	383	8	448	9	383	8
	bccm-IF-6C	367	8	415	10	389	9	483	10	437	9	486	11	439	10	486	11	441	10
	bccm-IF-7A	444	11	469	12	445	11	525	13	460	11	620	16	527	14	620	16	527	14
	bccm-IF-7B	444	11	469	12	446	11	525	13	477	12	620	16	523	13	620	16	523	13
	bccm-IF-7C	444	11	469	12	452	11	529	13	481	12	620	16	527	14	620	16	527	14
	bccm-IF-8A	545	9	676	11	630	10	745	11	675	10	987	16	917	14	987	16	917	14
	bccm-IF-8B	545	9	676	11	624	10	745	11	673	10	987	16	876	14	987	16	876	14
	bccm-IF-8C	547	10	676	11	659	10	748	11	732	11	1007	16	909	14	1007	16	909	14
	bccm-IF-9A	634	14	699	17	669	15	748	17	667	15	1082	26	976	23	1082	26	976	23
	bccm-IF-9B	546	14	699	17	657	15	748	17	670	15	1082	26	933	22	1082	26	933	22
	bccm-IF-9C	622	14	699	17	653	15	748	17	670	15	1082	26	963	22	1082	26	963	22
	bccm-IF-9D	546	14	684	16	670	15	719	16	670	15	1082	26	955	22	1082	26	955	22
	bccm-IF-10A	731	13	809	15	784	14	801	14	761	13	1179	21	1095	20	1179	21	1095	20
	bccm-IF-10B	732	13	809	15	783	14	801	14	759	13	1179	21	1034	18	1179	21	1034	18
	bccm-IF-10C	740	13	809	15	782	14	801	14	756	13	1179	21	1085	19	1179	21	1085	19
	bccm-IF-10D	732	13	809	15	785	14	802	14	792	14	1179	21	1087	20	1179	21	1087	20
	Mean Z_{gap} and K_{gap}			18.1 %	1.9	9.9 %	0.7	25.4 %	1.6	12.3 %	0.5	68.6 %	6.9	48.0 %	4.6	68.5 %	6.9	47.7 %	4.6
min K	bccm-IF-1A	229	6	289	8	257	7	280	7	240	6	365	10	309	8	365	10	309	8
	bccm-IF-1B	229	6	289	8	258	7	280	7	240	6	365	10	309	8	365	10	309	8
	bccm-IF-1C	259	8	317	9	279	8	335	9	298	8	390	11	337	9	390	11	330	9
	bccm-IF-2A	434	6	480	7	434	6	507	7	441	6	743	11	687	9	743	11	687	9

bccm-IF-2B	434	6	480	7	436	6	507	7	441	6	743	11	681	9	743	11	681	9
bccm-IF-2C	488	7	549	8	533	8	554	8	553	8	863	13	754	10	863	13	754	10
bccm-IF-3A	120	5	131	5	125	5	145	6	128	5	174	7	153	6	174	7	153	6
bccm-IF-3B	120	5	131	5	125	5	134	5	127	5	174	7	157	6	174	7	157	6
bccm-IF-3C	126	5	159	7	139	6	151	6	143	6	201	8	173	7	199	8	164	6
bccm-IF-4A	533	8	612	8	589	8	769	10	622	8	957	13	767	10	957	13	767	10
bccm-IF-4B	533	8	612	8	603	8	702	9	626	8	957	13	812	10	957	13	812	10
bccm-IF-4C	533	8	612	8	592	8	766	10	625	8	957	13	790	10	957	13	790	10
bccm-IF-4D	538	9	692	10	629	9	697	9	689	9	974	14	839	11	974	14	842	11
bccm-IF-5A	879	13	1059	15	986	14	1025	14	1007	14	1848	27	1608	22	1848	27	1608	22
bccm-IF-5B	879	13	1059	15	991	14	1025	14	1006	14	1848	27	1599	22	1848	27	1599	22
bccm-IF-5C	879	13	1059	15	981	13	1025	14	1001	14	1848	27	1596	22	1848	27	1596	22
bccm-IF-5D	879	13	1059	15	989	14	1085	15	1009	14	1848	27	1616	22	1848	27	1616	22
bccm-IF-6A	343	7	377	8	347	7	410	8	357	7	448	9	387	8	448	9	387	8
bccm-IF-6B	343	7	377	8	345	7	410	8	359	7	448	9	383	8	448	9	383	8
bccm-IF-6C	367	8	430	9	389	9	473	10	433	9	486	11	439	9	486	11	441	9
bccm-IF-7A	444	11	469	12	445	11	525	13	457	11	620	16	527	13	620	16	527	13
bccm-IF-7B	444	11	469	12	446	11	525	13	465	11	620	16	523	13	620	16	523	13
bccm-IF-7C	444	11	469	12	452	11	526	13	476	12	620	16	527	13	620	16	527	13
bccm-IF-8A	545	9	666	10	630	9	742	11	675	10	987	16	917	13	987	16	917	13
bccm-IF-8B	545	9	666	10	624	10	742	11	670	10	987	16	876	13	987	16	876	13
bccm-IF-8C	547	10	670	10	659	10	688	10	701	10	1007	16	909	13	1007	16	909	13
bccm-IF-9A	634	14	685	16	669	15	713	16	664	15	1090	25	976	22	1090	25	976	22
bccm-IF-9B	546	14	685	16	657	15	713	16	667	15	1090	25	933	21	1090	25	933	21
bccm-IF-9C	622	14	685	16	653	14	713	16	667	15	1090	25	963	21	1090	25	963	21
bccm-IF-9D	546	14	684	16	670	15	719	16	667	15	1082	26	955	21	1082	26	955	21
bccm-IF-10A	731	13	786	14	766	14	760	13	755	13	1179	21	1095	19	1179	21	1095	19
bccm-IF-10B	732	13	786	14	765	14	760	13	753	13	1179	21	1034	17	1179	21	1034	17
bccm-IF-10C	740	13	786	14	768	14	760	13	751	13	1179	21	1085	18	1179	21	1085	18
bccm-IF-10D	732	13	786	14	766	14	802	14	763	13	1179	21	1087	19	1179	21	1087	19
Mean Z_{gap} and K_{gap}			15.5 %	1.1	8.6 %	0.5	21.4 %	1.2	11.4 %	0.4	69.0 %	6.7	48.0 %	3.9	68.9 %	6.7	47.7 %	3.9

Table A.7: Heuristic results for *gdb-IF* set

		Best found		IM		RM(1000)		PS		PSRL(1000)		ERC		ERCRL(1000)		RC		RCRL(1000)	
1^{st} Obj	Instance	Z	K	Z	K	$\overline{Z}$	$\overline{K}$	Z	K	$\overline{Z}$	$\overline{K}$	Z	K	$\overline{Z}$	$\overline{K}$	Z	K	$\overline{Z}$	$\overline{K}$
min Z	gdb-IF-1	345	6	351	6	345	6	407	7	350	6	441	8	383	7	441	8	383	7
	gdb-IF-2	345	6	399	7	366	7	353	6	354	6	465	9	393	7	457	8	393	7
	gdb-IF-3	312	6	326	6	312	6	332	6	327	6	379	7	337	6	379	7	334	6
	gdb-IF-4	460	6	548	7	462	6	483	6	462	6	756	10	659	8	756	10	659	8
	gdb-IF-5	586	7	645	8	640	8	649	8	593	7	1012	13	841	10	1012	13	841	10
	gdb-IF-6	301	4	341	5	302	4	339	4	310	4	355	6	310	4	355	5	310	4
	gdb-IF-7	371	6	412	7	373	6	457	8	373	6	504	9	424	7	504	9	424	7
	gdb-IF-8	445	10	477	11	456	10	516	11	478	10	630	15	535	12	624	15	529	12
	gdb-IF-9	354	9	386	11	368	10	395	9	390	9	485	13	454	11	471	12	446	11
	gdb-IF-10	423	7	470	8	425	7	569	10	479	8	623	11	577	10	623	11	577	10
	gdb-IF-11	606	10	697	12	628	10	726	12	661	11	939	16	853	14	939	16	853	14
	gdb-IF-12	835	8	1092	10	903	9	941	9	846	8	1419	14	1211	12	1419	14	1211	12
	gdb-IF-13	602	5	634	6	618	6	700	6	644	6	714	7	652	7	712	7	652	7
	gdb-IF-14	138	8	179	10	138	8	141	8	139	8	209	12	178	10	209	12	178	10
	gdb-IF-15	60	3	64	5	60	3	64	4	60	3	66	4	64	4	66	4	64	4
	gdb-IF-16	156	12	157	12	157	12	156	12	156	12	195	16	172	13	195	16	172	13
	gdb-IF-17	99	8	105	9	99	8	105	9	101	8	117	10	109	9	117	10	109	9
	gdb-IF-18	222	10	268	13	241	12	244	11	228	10	330	16	315	15	330	16	315	15
	gdb-IF-19	83	3	91	3	83	3	91	3	83	3	93	3	83	3	93	3	83	3
	gdb-IF-20	179	7	191	8	179	7	191	7	180	7	231	9	203	8	231	9	203	8
	gdb-IF-21	190	12	202	13	192	12	210	13	194	12	232	16	213	14	232	16	213	14
	gdb-IF-22	265	19	285	20	265	19	288	20	276	19	316	23	298	21	316	23	298	21
	gdb-IF-23	282	20	299	23	286	21	302	22	295	21	343	28	317	24	343	28	317	24
	Mean Z_{gap} and K_{gap}			11.0 %	1.2	2.2 %	0.3	11.2 %	0.8	3.4 %	0.2	34.8 %	3.6	20.2 %	1.9	34.5 %	3.5	20 %	1.9
min K	gdb-IF-1	345	6	351	6	345	6	373	6	347	6	441	8	383	6	441	8	383	6
	gdb-IF-2	345	6	399	7	366	7	353	6	354	6	471	8	393	7	457	8	393	7
	gdb-IF-3	312	6	326	6	312	6	332	6	321	6	379	7	337	6	379	7	334	6
	gdb-IF-4	460	6	495	6	462	6	483	6	462	6	756	10	659	7	756	10	659	7
	gdb-IF-5	586	7	645	8	640	8	646	8	593	7	1012	13	841	10	1012	13	841	10
	gdb-IF-6	301	4	341	5	302	4	339	4	310	4	367	5	310	4	355	5	310	4
	gdb-IF-7	371	6	412	7	373	6	457	8	373	6	504	9	424	7	504	9	424	7

gdb-IF-8	445	10	472	10	454	10	492	10	474	10	634	14	535	11	628	14	529	11
gdb-IF-9	354	9	388	10	365	9	395	9	385	9	511	12	454	10	471	12	446	10
gdb-IF-10	423	7	470	8	425	7	524	9	477	8	623	11	577	10	623	11	577	10
gdb-IF-11	606	10	694	11	619	10	686	11	641	10	939	16	853	13	939	16	853	13
gdb-IF-12	835	8	1092	10	903	8	915	9	846	8	1419	14	1211	11	1419	14	1211	11
gdb-IF-13	602	5	634	6	617	6	700	6	640	6	714	7	652	6	721	6	652	6
gdb-IF-14	138	8	179	10	138	8	141	8	139	8	209	12	178	10	209	12	178	10
gdb-IF-15	60	3	64	4	60	3	64	4	60	3	66	4	64	4	66	4	64	4
gdb-IF-16	156	12	157	12	157	12	156	12	156	12	195	16	172	13	195	16	172	13
gdb-IF-17	99	8	105	8	99	8	103	8	101	8	117	10	109	8	117	10	109	8
gdb-IF-18	222	10	268	13	237	11	244	11	227	10	330	16	315	14	330	16	315	14
gdb-IF-19	83	3	91	3	83	3	91	3	83	3	93	3	83	3	93	3	83	3
gdb-IF-20	179	7	191	8	179	7	188	7	180	7	231	9	203	7	231	9	203	7
gdb-IF-21	190	12	202	13	191	12	207	13	193	12	232	16	214	13	232	16	214	13
gdb-IF-22	265	19	285	20	265	19	286	20	274	19	316	23	298	20	316	23	298	20
gdb-IF-23	282	20	295	22	285	20	302	22	293	21	343	28	317	23	343	28	317	23
Mean Z_{gap} and K_{gap}			10.4 %	0.9	1.9 %	0.2	9.3 %	0.6	2.9 %	0.1	35.5 %	3.4	20.2 %	1.3	34.6 %	3.4	20 %	1.3

Table A.8: Computational times, in seconds, of heuristics for *mval-IF-3L*.

Instance	α	IM	RM	PS	PSRL	ERC	ERCRL	U	RCRL
mval-IF-3L-1A	1000	0.1	61.2	0.0	4.2	0.0	3.4	0.1	12.2
mval-IF-3L-1B	1000	0.1	56.1	0.0	4.7	0.0	5.2	0.0	8.2
mval-IF-3L-1C	1000	0.1	53.3	0.0	3.8	0.0	3.5	0.0	6.1
mval-IF-3L-2A	1000	0.0	41.6	0.0	3.1	0.0	2.7	0.0	6.8
mval-IF-3L-2B	1000	0.0	44.6	0.0	2.3	0.0	3.4	0.1	9.7
mval-IF-3L-2C	1000	0.0	44.0	0.0	2.3	0.0	4.4	0.0	6.0
mval-IF-3L-3A	1000	0.0	46.3	0.0	2.1	0.0	2.4	0.1	8.7
mval-IF-3L-3B	1000	0.0	38.8	0.0	3.8	0.0	3.4	0.0	7.7
mval-IF-3L-3C	1000	0.0	38.2	0.0	4.0	0.0	4.6	0.0	6.6
mval-IF-3L-4A	1000	0.2	162.5	0.1	7.8	0.1	12.7	0.3	42.1
mval-IF-3L-4B	1000	0.2	175.7	0.0	4.6	0.1	8.6	0.2	33.3
mval-IF-3L-4C	1000	0.2	171.5	0.0	4.3	0.1	9.1	0.2	27.3
mval-IF-3L-4D	1000	0.2	169.4	0.0	5.4	0.1	10.4	0.1	19.3
mval-IF-3L-5A	1000	0.2	191.8	0.0	4.5	0.0	7.5	0.2	26.5
mval-IF-3L-5B	1000	0.3	299.6	0.0	3.5	0.0	7.5	0.1	21.3
mval-IF-3L-5C	1000	0.3	251.5	0.0	4.5	0.1	9.2	0.1	23.6
mval-IF-3L-5D	1000	0.3	308.4	0.0	4.0	0.1	8.5	0.1	20.0
mval-IF-3L-6A	1000	0.1	149.3	0.0	3.3	0.0	4.7	0.1	15.2
mval-IF-3L-6B	1000	0.1	136.7	0.0	2.6	0.0	5.4	0.1	13.5
mval-IF-3L-6C	1000	0.2	161.3	0.1	6.4	0.0	5.1	0.0	8.3
mval-IF-3L-7A	1000	0.3	281.8	0.0	3.0	0.1	9.2	0.1	20.8
mval-IF-3L-7B	1000	0.3	268.6	0.0	3.5	0.0	6.7	0.1	18.2
mval-IF-3L-7C	1000	0.3	273.0	0.1	12.1	0.1	10.0	0.1	15.2
mval-IF-3L-8A	1000	0.2	175.6	0.0	4.4	0.0	6.8	0.2	28.6
mval-IF-3L-8B	1000	0.2	172.3	0.0	6.2	0.0	6.8	0.2	25.3
mval-IF-3L-8C	1000	0.1	136.3	0.0	4.5	0.0	7.7	0.1	10.7
mval-IF-3L-9A	1000	0.3	314.8	0.0	6.2	0.1	11.5	0.3	44.4
mval-IF-3L-9B	1000	0.3	337.9	0.0	6.0	0.1	10.5	0.2	33.1
mval-IF-3L-9C	1000	0.3	327.1	0.1	13.7	0.1	11.0	0.2	35.3
mval-IF-3L-9D	1000	0.3	313.7	0.1	7.2	0.1	14.2	0.2	30.0
mval-IF-3L-10A	1000	0.3	333.1	0.1	15.3	0.1	22.8	0.3	57.7
mval-IF-3L-10B	1000	0.3	316.8	0.0	6.2	0.2	28.2	0.4	59.4
mval-IF-3L-10C	1000	0.3	337.6	0.1	12.2	0.1	24.1	0.4	61.9
mval-IF-3L-10D	1000	0.3	320.1	0.1	6.4	0.1	18.4	0.2	38.7
Mean time		0.2	191.5	0.0	5.5	0.1	9.1	0.1	23.6

Table A.9: Computational times, in seconds, of heuristics for *bccm-IF-3L*.

Instance	α	IM	RM	PS	PSRL	ERC	ERCRL	RC	RCRL
bccm-IF-3L-1A	1000	0.1	63.6	0.0	1.6	0.0	2.1	0.0	5.2
bccm-IF-3L-1B	1000	0.1	69.3	0.0	1.5	0.0	2.7	0.0	4.5
bccm-IF-3L-1C	1000	0.1	65.7	0.0	2.7	0.0	3.1	0.0	3.7
bccm-IF-3L-2A	1000	0.0	49.4	0.0	1.7	0.0	2.3	0.0	5.1
bccm-IF-3L-2B	1000	0.1	56.7	0.0	1.7	0.0	2.1	0.0	4.3
bccm-IF-3L-2C	1000	0.1	50.9	0.0	1.6	0.0	4.3	0.0	3.7
bccm-IF-3L-3A	1000	0.1	54.4	0.0	3.6	0.0	2.1	0.0	5.4
bccm-IF-3L-3B	1000	0.1	51.3	0.0	2.3	0.0	2.2	0.0	4.7
bccm-IF-3L-3C	1000	0.1	50.3	0.0	2.4	0.0	2.4	0.0	3.2
bccm-IF-3L-4A	1000	0.2	216.3	0.1	7.7	0.0	8.0	0.1	22.5
bccm-IF-3L-4B	1000	0.2	220.1	0.0	3.1	0.0	8.1	0.1	19.5
bccm-IF-3L-4C	1000	0.2	213.8	0.1	7.1	0.0	6.3	0.1	15.8
bccm-IF-3L-4D	1000	0.2	226.7	0.0	3.3	0.0	5.3	0.1	10.4
bccm-IF-3L-5A	1000	0.2	192.7	0.0	3.6	0.0	3.3	0.1	13.2
bccm-IF-3L-5B	1000	0.2	193.0	0.0	4.3	0.0	6.0	0.1	12.1

bccm-IF-3L-5C	1000	0.2	199.9	0.0	4.3	0.0	4.2	0.1	9.7
bccm-IF-3L-5D	1000	0.2	188.5	0.0	3.4	0.0	6.5	0.1	9.9
bccm-IF-3L-6A	1000	0.1	108.6	0.0	2.2	0.0	6.3	0.1	11.5
bccm-IF-3L-6B	1000	0.1	117.0	0.0	3.9	0.0	4.6	0.1	9.2
bccm-IF-3L-6C	1000	0.1	119.8	0.0	2.7	0.0	5.3	0.0	6.0
bccm-IF-3L-7A	1000	0.2	192.5	0.0	2.7	0.0	6.3	0.1	13.7
bccm-IF-3L-7B	1000	0.2	201.6	0.0	5.5	0.0	6.2	0.1	15.2
bccm-IF-3L-7C	1000	0.2	201.1	0.0	3.6	0.0	6.2	0.1	8.4
bccm-IF-3L-8A	1000	0.2	182.0	0.0	5.3	0.0	5.6	0.1	14.5
bccm-IF-3L-8B	1000	0.2	181.4	0.0	3.0	0.0	4.7	0.1	11.8
bccm-IF-3L-8C	1000	0.2	179.6	0.0	3.1	0.0	4.9	0.0	6.8
bccm-IF-3L-9A	1000	0.4	368.2	0.0	4.0	0.1	14.3	0.2	36.7
bccm-IF-3L-9B	1000	0.4	413.4	0.1	12.7	0.0	6.9	0.2	30.2
bccm-IF-3L-9C	1000	0.5	540.4	0.1	7.8	0.1	10.7	0.2	35.1
bccm-IF-3L-9D	1000	0.4	394.8	0.0	6.2	0.1	13.1	0.1	18.5
bccm-IF-3L-10A	1000	0.4	426.9	0.1	7.6	0.1	14.4	0.2	37.6
bccm-IF-3L-10B	1000	0.4	413.9	0.1	8.3	0.1	9.3	0.2	35.4
bccm-IF-3L-10C	1000	0.4	428.9	0.1	7.1	0.1	8.4	0.2	27.1
bccm-IF-3L-10D	1000	0.4	437.3	0.0	4.7	0.0	8.1	0.1	14.2
Mean time		0.2	207.9	0.0	4.3	0.0	6.1	0.1	14.3

Table A.10: Computational times, in seconds, of heuristics for *gdb-IF-3L*.

Instance	α	IM	RM	PS	PSRL	ERC	ERCRL	RC	RCRL
gdb-IF-3L-1	1000	0.0	19.5	0.0	1.2	0.0	2.4	0.0	2.2
gdb-IF-3L-2	1000	0.0	27.8	0.0	2.5	0.0	2.6	0.0	3.3
gdb-IF-3L-3	1000	0.0	20.4	0.0	1.7	0.0	2.3	0.0	2.4
gdb-IF-3L-4	1000	0.0	16.4	0.0	1.3	0.0	1.5	0.0	1.8
gdb-IF-3L-5	1000	0.0	28.4	0.0	3.6	0.0	3.4	0.0	3.9
gdb-IF-3L-6	1000	0.0	20.8	0.0	1.5	0.0	1.5	0.0	1.8
gdb-IF-3L-7	1000	0.0	23.7	0.0	1.2	0.0	1.8	0.0	2.2
gdb-IF-3L-8	1000	0.1	95.8	0.0	3.8	0.0	5.0	0.0	6.6
gdb-IF-3L-9	1000	0.1	117.6	0.0	3.2	0.0	6.2	0.0	6.1
gdb-IF-3L-10	1000	0.0	27.9	0.0	3.3	0.0	2.3	0.0	3.1
gdb-IF-3L-11	1000	0.1	94.8	0.0	3.1	0.0	4.1	0.0	5.3
gdb-IF-3L-12	1000	0.0	24.2	0.0	3.5	0.0	2.1	0.0	2.3
gdb-IF-3L-13	1000	0.0	29.2	0.0	1.9	0.0	2.4	0.0	2.4
gdb-IF-3L-14	1000	0.0	18.5	0.0	2.4	0.0	1.8	0.0	2.3
gdb-IF-3L-15	1000	0.0	19.2	0.0	2.1	0.0	1.5	0.0	1.9
gdb-IF-3L-16	1000	0.0	32.2	0.0	1.8	0.0	2.3	0.0	2.6
gdb-IF-3L-17	1000	0.0	34.4	0.0	1.7	0.0	1.9	0.0	2.4
gdb-IF-3L-18	1000	0.1	59.1	0.0	1.9	0.0	2.6	0.0	3.4
gdb-IF-3L-19	1000	0.0	5.4	0.0	1.7	0.0	1.0	0.0	1.0
gdb-IF-3L-20	1000	0.0	21.5	0.0	3.0	0.0	2.1	0.0	2.4
gdb-IF-3L-21	1000	0.0	43.4	0.0	2.2	0.0	2.7	0.0	3.2
gdb-IF-3L-22	1000	0.1	94.4	0.0	3.0	0.0	5.7	0.0	6.4
gdb-IF-3L-23	1000	0.1	136.7	0.1	10.8	0.0	6.9	0.1	9.3
Mean time		0.0	44.0	0.0	2.7	0.0	2.9	0.0	3.4

Table A.11: Computational times, in seconds, of heuristics for *bccm-IF*.

Instance	α	IM	RM	PS	PSRL	ERC	ERCRL	RC	RCRL
bccm-IF-1A	1000	0.1	70.2	0.0	4.3	0.0	5.2	0.0	5.6
bccm-IF-1B	1000	0.1	69.5	0.0	5.8	0.0	4.9	0.0	5.5
bccm-IF-1C	1000	0.1	68.7	0.0	5.3	0.0	5.9	0.0	6.3

bccm-IF-2A	1000	0.1	63.5	0.0	3.3	0.0	4.1	0.0	4.6
bccm-IF-2B	1000	0.1	51.6	0.0	4.1	0.0	5.5	0.0	6.1
bccm-IF-2C	1000	0.1	56.2	0.1	7.6	0.0	4.7	0.0	5.0
bccm-IF-3A	1000	0.1	58.6	0.0	3.8	0.0	3.4	0.0	4.0
bccm-IF-3B	1000	0.1	56.2	0.0	2.4	0.0	4.5	0.0	4.9
bccm-IF-3C	1000	0.1	55.0	0.0	3.3	0.0	3.8	0.0	4.2
bccm-IF-4A	1000	0.2	223.3	0.0	4.7	0.1	10.8	0.1	12.5
bccm-IF-4B	1000	0.2	234.3	0.1	8.3	0.1	10.5	0.1	12.6
bccm-IF-4C	1000	0.2	222.3	0.1	11.5	0.1	10.2	0.1	11.8
bccm-IF-4D	1000	0.2	227.3	0.1	6.5	0.0	8.1	0.1	9.7
bccm-IF-5A	1000	0.2	203.1	0.2	22.1	0.1	8.7	0.1	9.5
bccm-IF-5B	1000	0.2	199.9	0.1	10.3	0.0	8.2	0.1	9.3
bccm-IF-5C	1000	0.2	209.6	0.1	14.0	0.1	8.8	0.1	9.6
bccm-IF-5D	1000	0.2	198.4	0.1	14.1	0.0	7.9	0.1	8.8
bccm-IF-6A	1000	0.1	116.6	0.0	4.9	0.0	7.6	0.1	9.0
bccm-IF-6B	1000	0.1	127.6	0.0	3.8	0.0	7.5	0.1	8.5
bccm-IF-6C	1000	0.1	118.0	0.1	11.9	0.1	8.8	0.1	9.6
bccm-IF-7A	1000	0.2	203.7	0.1	10.9	0.1	10.3	0.1	11.6
bccm-IF-7B	1000	0.2	219.9	0.1	10.8	0.1	10.5	0.1	11.3
bccm-IF-7C	1000	0.2	211.4	0.1	18.2	0.1	10.4	0.1	11.3
bccm-IF-8A	1000	0.2	189.5	0.1	10.6	0.0	6.8	0.0	8.1
bccm-IF-8B	1000	0.2	194.9	0.1	12.0	0.0	7.0	0.0	8.2
bccm-IF-8C	1000	0.2	180.8	0.1	14.4	0.0	7.8	0.1	8.8
bccm-IF-9A	1000	0.5	492.4	0.2	20.9	0.1	11.9	0.1	13.5
bccm-IF-9B	1000	0.6	636.7	0.1	18.3	0.1	11.4	0.1	12.8
bccm-IF-9C	1000	0.4	392.1	0.2	30.3	0.1	11.6	0.1	13.2
bccm-IF-9D	1000	0.4	396.7	0.2	26.3	0.1	11.9	0.1	13.9
bccm-IF-10A	1000	0.7	745.7	0.2	20.2	0.1	9.6	0.1	11.7
bccm-IF-10B	1000	0.7	685.7	0.2	22.3	0.1	9.9	0.1	11.7
bccm-IF-10C	1000	0.4	435.4	0.1	11.4	0.1	9.4	0.1	11.4
bccm-IF-10D	1000	0.4	431.2	0.1	18.7	0.1	10.3	0.1	11.7
Mean time		0.2	236.6	0.1	11.7	0.0	8.2	0.1	9.3

Table A.12: Computational times, in seconds, of heuristics for *gdb-IF*.

Instance	α	IM	RM	PS	PSRL	ERC	ERCRL	RC	RCRL
gdb-IF-1	1000	0.0	24.1	0.0	2.4	0.0	3.4	0.0	3.5
gdb-IF-2	1000	0.0	34.7	0.0	4.1	0.0	4.2	0.0	4.8
gdb-IF-3	1000	0.0	24.4	0.0	4.9	0.0	3.6	0.0	3.9
gdb-IF-4	1000	0.0	19.6	0.0	3.5	0.0	4.2	0.0	4.3
gdb-IF-5	1000	0.0	37.3	0.1	7.9	0.0	4.5	0.0	4.5
gdb-IF-6	1000	0.0	21.6	0.0	2.8	0.0	2.5	0.0	2.7
gdb-IF-7	1000	0.0	23.5	0.1	7.8	0.0	5.2	0.0	5.5
gdb-IF-8	1000	0.1	110.5	0.1	11.6	0.0	5.0	0.0	6.1
gdb-IF-9	1000	0.2	164.6	0.0	5.5	0.0	5.6	0.1	8.4
gdb-IF-10	1000	0.0	28.5	0.0	4.9	0.0	4.6	0.0	4.6
gdb-IF-11	1000	0.1	108.3	0.0	5.6	0.0	6.5	0.0	7.1
gdb-IF-12	1000	0.0	30.3	0.0	6.1	0.0	4.6	0.0	4.7
gdb-IF-13	1000	0.0	34.2	0.0	6.1	0.0	4.9	0.0	4.3
gdb-IF-14	1000	0.0	26.9	0.0	4.8	0.0	5.1	0.0	4.9
gdb-IF-15	1000	0.0	17.9	0.0	1.5	0.0	1.4	0.0	1.6
gdb-IF-16	1000	0.0	46.2	0.1	11.9	0.0	7.3	0.0	7.2
gdb-IF-17	1000	0.0	37.9	0.0	5.3	0.0	3.5	0.0	3.7
gdb-IF-18	1000	0.1	70.4	0.1	11.2	0.0	5.5	0.0	5.6
gdb-IF-19	1000	0.0	5.7	0.0	1.2	0.0	1.6	0.0	1.7
gdb-IF-20	1000	0.0	24.2	0.0	5.0	0.0	4.4	0.0	4.5
gdb-IF-21	1000	0.1	55.4	0.1	9.5	0.0	6.3	0.0	6.5
gdb-IF-22	1000	0.1	114.5	0.1	11.3	0.0	6.0	0.0	6.3

gdb-IF-23	1000	0.2	158.0	0.2	30.6	0.1	8.8	0.1	9.7
Mean time		0.1	53.0	0.1	7.2	0.0	4.7	0.0	5.0

A.2 Full results for the tabu search metaheuristic

This section contains full result tables, referred to in Chapter 7, for the Neutral Accelerated Tabu Search metaheuristic.

A.2.1 Results on *Act-IF*, *Cen-IF*, *Lpr-IF* and *mval-IF-3L*

In this subsection results are shown for the following heuristic setups: Multi-start Randomised Constructive Heuristic (M-CH); Deterministic Local Search Accelerated-Extended-Greedy (D-LS); Multi-start Local Search Accelerated-Extended-Greedy (M-LS); and Neutral-Accelerated Tabu Search (NATS). All the setups were tested with the following constructive heuristics: *Efficient-Route-Cluster* (ERC); *Path-Scanning* (PS); and *Merge* (M). Solution cost results for the setups are shown in Table A.13, and vehicle fleet size results are shown in Table A.14. The results are summarised in Tables 7.1 and 7.2 in Chapter 7.

Table A.13: Expected cost, Z, for the constructive heuristic, accelerated Local Search, and Neutral Accelerated Tabu Search setups under different time-limits on the *Cen-IF*, *Lpr-IF*, *Act-IF*, and *mval-IF-3L* sets.

		3 minute time-limit				30 minute time-limit				60 minute time-limit			
Instance	Construct	M-CH	D-LS	M-LS	NATS	M-CH	D-LS	M-LS	NATS	M-CH	D-LS	M-LS	NATS
Cen-IF-a	ERC	24033643	23411100	23584719	23262900	239939	234111	234085	231205	239939	234111	234085	230813
Cen-IF-a	PS	24818280	24031300	23897837	23732600	247214	240313	237646	235445	247214	240313	237646	235403
Cen-IF-a	M	-	-	-	-	233728	232868	231544	231114	233728	232868	231544	230998
Cen-IF-b	ERC	60337412	58451400	59884614	58451400	601158	584514	588237	580050	601158	584514	588237	578098
Cen-IF-b	PS	61164996	58698300	59984126	59680000	609561	586983	587751	581087	609561	586983	587751	578601
Cen-IF-b	M	-	-	-	-	608468	592205	591971	588898	608468	592205	591971	583936
Cen-IF-c	ERC	53227870	52114300	52958534	52114300	530889	521143	519464	516139	530889	521143	519464	514179
Cen-IF-c	PS	54837915	52753700	53762008	52753700	544847	527537	524012	521997	544847	527537	524012	520612
Cen-IF-c	M	-	-	-	-	527485	514852	513461	511569	527485	514852	513461	510197
Lpr-IF-a-01	ERC	1359400	1360900	1352300	1356400	13594	13609	13523	13564	13594	13609	13523	13564
Lpr-IF-a-01	PS	1366700	1358900	1352600	1358900	13667	13589	13526	13589	13667	13589	13526	13589
Lpr-IF-a-01	M	1382000	1368600	1352300	1361500	13820	13686	13523	13615	13820	13686	13523	13615
Lpr-IF-a-02	ERC	2843700	2837700	2821500	2824600	28437	28377	28215	28246	28437	28377	28215	28246
Lpr-IF-a-02	PS	2867900	2863500	2820200	2824300	28679	28635	28202	28243	28679	28635	28202	28243
Lpr-IF-a-02	M	2859388	2834600	2826446	2823200	28586	28346	28250	28232	28586	28346	28250	28232
Lpr-IF-a-03	ERC	7881300	7798800	7765252	7723300	78813	77988	77496	77214	78813	77988	77496	77214
Lpr-IF-a-03	PS	7950700	7815400	7772843	7711400	79507	78154	77612	77114	79507	78154	77612	77114
Lpr-IF-a-03	M	7848945	7815900	7768119	7722400	78161	78159	77534	77109	78161	78159	77534	77109
Lpr-IF-a-04	ERC	13310400	13181800	13118764	13001200	133104	131818	130989	129867	133104	131818	130989	129867
Lpr-IF-a-04	PS	13342600	13118500	13122663	13015700	133426	131185	131048	129954	133426	131185	131048	129954
Lpr-IF-a-04	M	14831969	13310900	13251175	13023800	147133	133109	132069	129962	147133	133109	132069	129962
Lpr-IF-a-05	ERC	21142400	20895000	20835738	20640200	211424	208950	208013	205776	211424	208950	208013	205478
Lpr-IF-a-05	PS	21173100	20811100	20837468	20633700	211731	208111	208077	206068	211731	208111	208077	205870
Lpr-IF-a-05	M	22645953	21005700	21037759	20724700	226460	210057	209475	206388	226460	210057	209475	206254
Lpr-IF-b-01	ERC	1483500	1487600	1483500	1483900	14835	14876	14835	14839	14835	14876	14835	14839
Lpr-IF-b-01	PS	1483500	1487500	1483500	1483900	14835	14875	14835	14839	14835	14875	14835	14839
Lpr-IF-b-01	M	1506800	1487000	1483500	1483500	15068	14870	14835	14835	15068	14870	14835	14835
Lpr-IF-b-02	ERC	2894200	2893700	2872900	2872000	28942	28937	28729	28720	28942	28937	28729	28720
Lpr-IF-b-02	PS	2963100	2930900	2870200	2902000	29631	29309	28702	29020	29631	29309	28702	29020
Lpr-IF-b-02	M	2893900	2884000	2869615	2872000	28939	28840	28687	28720	28939	28840	28687	28720
Lpr-IF-b-03	ERC	7997700	7951100	7877869	7835600	79977	79511	78664	78142	79977	79511	78664	78142

Lpr-IF-b-03	PS	8041500	7943300	7893573	7844900	80415	79433	78871	78277	80415	79433	78871	78277
Lpr-IF-b-03	M	8100210	7943800	7968199	7868900	80574	79438	79409	78624	80574	79438	79409	78624
Lpr-IF-b-04	ERC	13231100	13125400	12977640	12910900	132311	131254	129591	128808	132311	131254	129591	128808
Lpr-IF-b-04	PS	13325200	13097600	13049569	12949600	133252	130976	130142	129254	133252	130976	130142	129254
Lpr-IF-b-04	M	14109176	13314700	13284024	13006000	139574	133147	132074	129260	139574	133147	132074	129225
Lpr-IF-b-05	ERC	22055500	21754700	21711695	21534300	220555	217547	216697	214647	220555	217547	216697	214430
Lpr-IF-b-05	PS	22210500	21762600	21738801	21502100	222105	217626	217119	214807	222105	217626	217119	214773
Lpr-IF-b-05	M	23278332	22083100	22134278	21671700	230891	220831	220488	215538	230891	220831	220488	215538
Lpr-IF-c-01	ERC	1873500	1877300	1870400	1870600	18735	18773	18704	18706	18735	18773	18704	18706
Lpr-IF-c-01	PS	1876500	1880300	1871700	1873600	18765	18803	18717	18736	18765	18803	18717	18736
Lpr-IF-c-01	M	1880300	1886600	1870400	1873700	18803	18866	18704	18737	18803	18866	18704	18737
Lpr-IF-c-02	ERC	3668900	3664400	3637900	3643300	36689	36644	36379	36433	36689	36644	36379	36433
Lpr-IF-c-02	PS	3668800	3651000	3639200	3639300	36688	36510	36392	36393	36688	36510	36392	36393
Lpr-IF-c-02	M	3718583	3672700	3648130	3643300	37168	36727	36458	36433	37168	36727	36458	36433
Lpr-IF-c-03	ERC	11417000	11353200	11293938	11239100	114170	113532	112797	112144	114170	113532	112797	112144
Lpr-IF-c-03	PS	11413700	11284700	11278174	11183400	114137	112847	112601	111793	114137	112847	112601	111793
Lpr-IF-c-03	M	11750708	11358300	11387859	11242500	116692	113583	113495	112336	116692	113583	113495	112336
Lpr-IF-c-04	ERC	17360400	17218500	17216930	17051800	173604	172185	171859	170045	173604	172185	171859	170045
Lpr-IF-c-04	PS	17625800	17372900	17300649	17150700	176258	173729	172716	171011	176258	173729	172716	170873
Lpr-IF-c-04	M	17457400	17290500	17233073	17137700	174000	172905	171956	170728	174000	172905	171956	170728
Lpr-IF-c-05	ERC	27380410	27152900	27018341	26773400	273802	271529	269521	265892	273802	271529	269521	265716
Lpr-IF-c-05	PS	27633438	27037600	27027203	26695600	276318	270376	269831	265260	276318	270376	269831	265009
Lpr-IF-c-05	M	28313355	27104800	27179909	26768400	282373	271048	270684	266179	282373	271048	270684	265903
Act-IF-a	ERC	2242400	2235300	2211866	2221000	22424	22353	22094	22210	22424	22353	22094	22210
Act-IF-a	PS	2236600	2227700	2220244	2224900	22366	22277	22168	22249	22366	22277	22168	22249
Act-IF-a	M	2635124	2251900	2307005	2239500	26218	22519	22831	22395	26218	22519	22831	22395
Act-IF-b	ERC	7251700	7201500	7170806	7155700	72517	72015	71590	71557	72517	72015	71590	71557
Act-IF-b	PS	7268300	7200200	7159985	7125000	72683	72002	71458	71250	72683	72002	71458	71250
Act-IF-b	M	8271009	7323800	7256126	7220400	81602	73238	72338	72049	81602	73238	72338	72049
Act-IF-c	ERC	5008100	4998400	4962054	4976100	50081	49984	49570	49761	50081	49984	49570	49761
Act-IF-c	PS	4997500	4975400	4960447	4963400	49975	49754	49531	49634	49975	49754	49531	49634
Act-IF-c	M	5377832	5029700	4998175	4971700	53476	50297	49841	49717	53476	50297	49841	49717
mval-IF-3L-1A	ERC	26800	25000	25000	25000	268	250	250	250	268	250	250	250
mval-IF-3L-1A	PS	25000	27700	25000	25900	250	277	250	259	250	277	250	259
mval-IF-3L-1A	M	28100	27800	25900	25900	281	278	259	259	281	278	259	259
mval-IF-3L-1B	ERC	33000	32500	30800	30900	330	325	308	309	330	325	308	309

mval-IF-3L-1B	PS	33200	32800	30800	30900	332	328	308	309	332	328	308	309
mval-IF-3L-1B	M	31500	32100	30700	30900	315	321	307	309	315	321	307	309
mval-IF-3L-1C	ERC	38600	38000	33900	35400	386	380	339	354	386	380	339	354
mval-IF-3L-1C	PS	38700	38700	33000	31800	387	387	330	318	387	387	330	318
mval-IF-3L-1C	M	36100	40100	35000	36900	361	401	350	369	361	401	350	369
mval-IF-3L-2A	ERC	49300	50100	41300	41300	493	501	413	413	493	501	413	413
mval-IF-3L-2A	PS	41300	42300	41300	41300	413	423	413	413	413	423	413	413
mval-IF-3L-2A	M	42500	41300	41300	41300	425	413	413	413	425	413	413	413
mval-IF-3L-2B	ERC	46900	43500	42300	43500	469	435	423	435	469	435	423	435
mval-IF-3L-2B	PS	43500	43700	42500	43700	435	437	425	437	435	437	425	437
mval-IF-3L-2B	M	42900	48500	42900	46900	429	485	429	469	429	485	429	469
mval-IF-3L-2C	ERC	53900	53500	48800	49200	539	535	488	492	539	535	488	492
mval-IF-3L-2C	PS	52100	48800	48100	46700	521	488	481	467	521	488	481	467
mval-IF-3L-2C	M	54500	56500	47900	48000	545	565	479	480	545	565	479	480
mval-IF-3L-3A	ERC	13900	14100	13500	13500	139	141	135	135	139	141	135	135
mval-IF-3L-3A	PS	13800	14200	13500	13600	138	142	135	136	138	142	135	136
mval-IF-3L-3A	M	13700	14300	13500	13500	137	143	135	135	137	143	135	135
mval-IF-3L-3B	ERC	15600	15800	14700	15000	156	158	147	150	156	158	147	150
mval-IF-3L-3B	PS	15100	17200	14700	14700	151	172	147	147	151	172	147	147
mval-IF-3L-3B	M	15500	16900	14700	16100	155	169	147	161	155	169	147	161
mval-IF-3L-3C	ERC	13100	13800	11700	11600	131	138	117	116	131	138	117	116
mval-IF-3L-3C	PS	13300	13300	11900	12200	133	133	119	122	133	133	119	122
mval-IF-3L-3C	M	13200	14200	11800	12000	132	142	118	120	132	142	118	120
mval-IF-3L-4A	ERC	73000	65300	64408	64400	730	653	644	644	730	653	644	644
mval-IF-3L-4A	PS	67600	67200	64611	64600	676	672	646	646	676	672	646	646
mval-IF-3L-4A	M	66100	68200	64401	64400	661	682	644	644	661	682	644	644
mval-IF-3L-4B	ERC	77800	75800	69914	73900	778	758	698	739	778	758	698	739
mval-IF-3L-4B	PS	77000	76100	73652	73900	770	761	736	739	770	761	736	739
mval-IF-3L-4B	M	78400	79200	71673	73900	784	792	711	739	784	792	711	739
mval-IF-3L-4C	ERC	78900	79800	72037	70000	789	798	716	700	789	798	716	700
mval-IF-3L-4C	PS	80800	78200	75032	73300	808	782	750	733	808	782	750	733
mval-IF-3L-4C	M	79600	78600	75337	73300	796	786	753	733	796	786	753	733
mval-IF-3L-4D	ERC	80700	85200	74466	74200	807	852	742	742	807	852	742	742
mval-IF-3L-4D	PS	82100	78900	76711	72200	821	789	766	722	821	789	766	722
mval-IF-3L-4D	M	79600	86400	75467	76900	796	864	753	769	796	864	753	769
mval-IF-3L-5A	ERC	81500	79900	77000	77600	815	799	770	776	815	799	770	776
mval-IF-3L-5A	PS	79200	81100	77400	77600	792	811	774	776	792	811	774	776
mval-IF-3L-5A	M	89929	77900	77750	76700	899	779	775	767	899	779	775	767

mval-IF-3L-5B	ERC	76500	73700	70941	71100	765	737	709	711	765	737	709	711
mval-IF-3L-5B	PS	76500	77500	71300	71500	765	775	713	715	765	775	713	715
mval-IF-3L-5B	M	83501	74100	71189	70500	820	741	709	705	820	741	709	705
mval-IF-3L-5C	ERC	91600	82100	79119	77500	916	821	791	775	916	821	791	775
mval-IF-3L-5C	PS	82900	84900	79114	78900	829	849	791	789	829	849	791	789
mval-IF-3L-5C	M	94771	89000	79260	82700	940	890	789	827	940	890	789	827
mval-IF-3L-5D	ERC	86600	89400	77381	75600	866	894	771	744	866	894	771	744
mval-IF-3L-5D	PS	84100	84100	78641	75200	841	841	786	752	841	841	786	752
mval-IF-3L-5D	M	94316	88800	78587	76300	942	888	782	763	942	888	782	763
mval-IF-3L-6A	ERC	37400	36000	34400	34200	374	360	344	342	374	360	344	342
mval-IF-3L-6A	PS	36600	37200	34900	34500	366	372	349	345	366	372	349	345
mval-IF-3L-6A	M	35600	36500	34518	34500	356	365	345	345	356	365	345	345
mval-IF-3L-6B	ERC	37000	35900	34200	34500	370	359	342	345	370	359	342	345
mval-IF-3L-6B	PS	38500	37100	35200	35900	385	371	352	359	385	371	352	359
mval-IF-3L-6B	M	34400	41200	33911	37500	344	412	339	375	344	412	339	375
mval-IF-3L-6C	ERC	47900	47700	42300	43600	479	477	423	436	479	477	423	436
mval-IF-3L-6C	PS	51100	49300	43800	48300	511	493	438	483	511	493	438	483
mval-IF-3L-6C	M	42600	45300	42124	41200	426	453	421	412	426	453	421	412
mval-IF-3L-7A	ERC	41600	39600	39400	38800	416	396	394	388	416	396	394	388
mval-IF-3L-7A	PS	43000	43000	40000	38800	430	430	400	388	430	430	400	388
mval-IF-3L-7A	M	39212	40800	38800	38800	390	408	388	388	390	408	388	388
mval-IF-3L-7B	ERC	49900	46200	45000	44200	499	462	450	442	499	462	450	442
mval-IF-3L-7B	PS	49100	48500	45610	43700	491	485	456	437	491	485	456	437
mval-IF-3L-7B	M	45567	49100	44239	44200	453	491	440	442	453	491	440	442
mval-IF-3L-7C	ERC	53100	51700	48300	48000	531	517	483	480	531	517	483	480
mval-IF-3L-7C	PS	56200	54100	50417	49000	562	541	504	490	562	541	504	490
mval-IF-3L-7C	M	50449	53200	49007	50100	503	532	488	501	503	532	488	501
mval-IF-3L-8A	ERC	69700	68400	65300	64400	697	684	653	644	697	684	653	644
mval-IF-3L-8A	PS	63400	65000	63400	64400	634	650	634	644	634	650	634	644
mval-IF-3L-8A	M	68400	67000	65048	66400	684	670	649	664	684	670	649	664
mval-IF-3L-8B	ERC	63600	66100	57000	57200	636	661	570	572	636	661	570	572
mval-IF-3L-8B	PS	60900	60000	58100	58200	609	600	581	582	609	600	581	582
mval-IF-3L-8B	M	61200	63500	58891	59500	612	635	588	595	612	635	588	595
mval-IF-3L-8C	ERC	69200	68900	60369	57400	692	689	603	574	692	689	603	574
mval-IF-3L-8C	PS	69500	69000	62600	60100	695	690	626	601	695	690	626	601
mval-IF-3L-8C	M	66100	68400	61141	58700	661	684	608	587	661	684	608	587
mval-IF-3L-9A	ERC	61200	61100	53072	52100	612	611	529	521	612	611	529	521
mval-IF-3L-9A	PS	59300	58000	53279	52300	593	580	531	523	593	580	531	523

mval-IF-3L-9A	M	59874	58300	54384	55400	591	583	534	554	591	583	534	554
mval-IF-3L-9B	ERC	58200	53200	51387	50500	582	532	512	505	582	532	512	505
mval-IF-3L-9B	PS	53700	54100	51506	50900	537	541	515	509	537	541	515	509
mval-IF-3L-9B	M	59426	55100	50700	50000	592	551	504	500	592	551	504	500
mval-IF-3L-9C	ERC	56900	56200	52555	52200	569	562	523	522	569	562	523	522
mval-IF-3L-9C	PS	53900	56400	51588	52700	539	564	515	523	539	564	515	523
mval-IF-3L-9C	M	59894	56200	53675	51900	595	562	522	519	595	562	522	519
mval-IF-3L-9D	ERC	71100	64700	63171	59400	711	647	630	584	711	647	630	584
mval-IF-3L-9D	PS	66600	67700	62926	58900	666	677	624	589	666	677	624	589
mval-IF-3L-9D	M	65711	65900	61797	59100	654	659	612	591	654	659	612	591
mval-IF-3L-10A	ERC	86200	84100	81172	80300	862	841	811	803	862	841	811	803
mval-IF-3L-10A	PS	84300	83700	81414	80400	843	837	813	804	843	837	813	804
mval-IF-3L-10A	M	89545	84400	80969	80300	877	844	808	801	877	844	808	801
mval-IF-3L-10B	ERC	85000	82000	79840	80000	850	820	798	792	850	820	798	792
mval-IF-3L-10B	PS	83000	81500	79832	79000	830	815	798	790	830	815	798	790
mval-IF-3L-10B	M	86792	83500	80619	79200	865	835	804	792	865	835	804	792
mval-IF-3L-10C	ERC	80000	76100	74341	73300	800	761	742	733	800	761	742	733
mval-IF-3L-10C	PS	80600	77600	74651	73400	806	776	741	728	806	776	741	728
mval-IF-3L-10C	M	81463	77000	74121	73900	811	770	738	732	811	770	738	732
mval-IF-3L-10D	ERC	82100	81000	76352	72000	821	810	761	720	821	810	761	720
mval-IF-3L-10D	PS	81600	81300	75275	72000	816	813	742	720	816	813	742	720
mval-IF-3L-10D	M	84532	82900	76283	71000	845	829	755	710	845	829	755	710

Table A.14: Expected fleet size, K, for the constructive heuristic, accelerated Local Search, and Neutral Accelerated Tabu Search setups under different time-limits on the *Cen-IF*, *Lpr-IF*, *Act-IF*, and *mval-IF-3L* sets.

		3 minute time-limit				30 minute time-limit				60 minute time-limit			
Instance	Construct	M-CH	D-LS	M-LS	NATS	M-CH	D-LS	M-LS	NATS	M-CH	D-LS	M-LS	NATS
Cen-IF-a	ERC	9	9	9	9	9	9	9	9	9	9	9	9
Cen-IF-a	PS	9	9	9	9	9	9	9	9	9	9	9	9
Cen-IF-a	M	-	-	-	-	9	9	9	9	9	9	9	9
Cen-IF-b	ERC	22	21	22	21	22	21	21	21	22	21	21	21
Cen-IF-b	PS	22	21	22	21	22	21	21	21	22	21	21	21
Cen-IF-b	M	-	-	-	-	27	21	21	21	27	21	21	21

Cen-IF-c	ERC	20	19	19	19	20	19	19	19	20	19	19	19
Cen-IF-c	PS	20	19	20	19	19	19	19	19	19	19	19	19
Cen-IF-c	M	-	-	-	-	19	19	19	19	19	19	19	19
Lpr-IF-a-01	ERC	1	1	1	1	1	1	1	1	1	1	1	1
Lpr-IF-a-01	PS	1	1	1	1	1	1	1	1	1	1	1	1
Lpr-IF-a-01	M	1	1	1	1	1	1	1	1	1	1	1	1
Lpr-IF-a-02	ERC	1	1	1	1	1	1	1	1	1	1	1	1
Lpr-IF-a-02	PS	1	1	1	1	1	1	1	1	1	1	1	1
Lpr-IF-a-02	M	1	1	1	1	1	1	1	1	1	1	1	1
Lpr-IF-a-03	ERC	3	3	3	3	3	3	3	3	3	3	3	3
Lpr-IF-a-03	PS	3	3	3	3	3	3	3	3	3	3	3	3
Lpr-IF-a-03	M	3	3	3	3	3	3	3	3	3	3	3	3
Lpr-IF-a-04	ERC	5	5	5	5	5	5	5	5	5	5	5	5
Lpr-IF-a-04	PS	5	5	5	5	5	5	5	5	5	5	5	5
Lpr-IF-a-04	M	6	5	5	5	6	5	5	5	6	5	5	5
Lpr-IF-a-05	ERC	8	8	8	8	8	8	8	8	8	8	8	8
Lpr-IF-a-05	PS	8	8	8	8	8	8	8	8	8	8	8	8
Lpr-IF-a-05	M	9	8	8	8	8	8	8	8	8	8	8	8
Lpr-IF-b-01	ERC	1	1	1	1	1	1	1	1	1	1	1	1
Lpr-IF-b-01	PS	1	1	1	1	1	1	1	1	1	1	1	1
Lpr-IF-b-01	M	1	1	1	1	1	1	1	1	1	1	1	1
Lpr-IF-b-02	ERC	2	2	1	2	2	2	1	2	2	2	1	2
Lpr-IF-b-02	PS	2	2	1	2	2	2	1	2	2	2	1	2
Lpr-IF-b-02	M	2	2	1	2	2	2	1	2	2	2	1	2
Lpr-IF-b-03	ERC	3	3	3	3	3	3	3	3	3	3	3	3
Lpr-IF-b-03	PS	3	3	3	3	3	3	3	3	3	3	3	3
Lpr-IF-b-03	M	3	3	3	3	3	3	3	3	3	3	3	3
Lpr-IF-b-04	ERC	5	5	5	5	5	5	5	5	5	5	5	5
Lpr-IF-b-04	PS	5	5	5	5	5	5	5	5	5	5	5	5
Lpr-IF-b-04	M	5	5	5	5	5	5	5	5	5	5	5	5
Lpr-IF-b-05	ERC	8	8	8	8	8	8	8	8	8	8	8	8
Lpr-IF-b-05	PS	8	8	8	8	8	8	8	8	8	8	8	8
Lpr-IF-b-05	M	9	8	8	8	9	8	8	8	9	8	8	8
Lpr-IF-c-01	ERC	1	1	1	1	1	1	1	1	1	1	1	1
Lpr-IF-c-01	PS	1	1	1	1	1	1	1	1	1	1	1	1
Lpr-IF-c-01	M	1	1	1	1	1	1	1	1	1	1	1	1
Lpr-IF-c-02	ERC	2	2	2	2	2	2	2	2	2	2	2	2

Lpr-IF-c-02	PS	2	2	2	2	2	2	2	2	2	2	2	2
Lpr-IF-c-02	M	2	2	2	2	2	2	2	2	2	2	2	2
Lpr-IF-c-03	ERC	4	4	4	4	4	4	4	4	4	4	4	4
Lpr-IF-c-03	PS	4	4	4	4	4	4	4	4	4	4	4	4
Lpr-IF-c-03	M	5	4	4	4	5	4	4	4	5	4	4	4
Lpr-IF-c-04	ERC	7	7	7	7	7	7	7	7	7	7	7	7
Lpr-IF-c-04	PS	7	7	7	6	7	7	7	6	7	7	7	6
Lpr-IF-c-04	M	7	7	7	6	7	7	7	6	7	7	7	6
Lpr-IF-c-05	ERC	10	10	10	10	10	10	10	10	10	10	10	10
Lpr-IF-c-05	PS	10	10	10	10	10	10	10	10	10	10	10	10
Lpr-IF-c-05	M	11	10	10	10	11	10	10	10	11	10	10	10
Act-IF-a	ERC	1	1	1	1	1	1	1	1	1	1	1	1
Act-IF-a	PS	1	1	1	1	1	1	1	1	1	1	1	1
Act-IF-a	M	1	1	1	1	1	1	1	1	1	1	1	1
Act-IF-b	ERC	3	3	3	3	3	3	3	3	3	3	3	3
Act-IF-b	PS	3	3	3	3	3	3	3	3	3	3	3	3
Act-IF-b	M	3	3	3	3	3	3	3	3	3	3	3	3
Act-IF-c	ERC	2	2	2	2	2	2	2	2	2	2	2	2
Act-IF-c	PS	2	2	2	2	2	2	2	2	2	2	2	2
Act-IF-c	M	2	2	2	2	2	2	2	2	2	2	2	2
mval-IF-3L-1A	ERC	3	2	2	2	3	2	2	2	3	2	2	2
mval-IF-3L-1A	PS	2	3	2	3	2	3	2	3	2	3	2	3
mval-IF-3L-1A	M	3	3	3	3	3	3	3	3	3	3	3	3
mval-IF-3L-1B	ERC	3	3	3	3	3	3	3	3	3	3	3	3
mval-IF-3L-1B	PS	3	3	3	3	3	3	3	3	3	3	3	3
mval-IF-3L-1B	M	3	3	3	3	3	3	3	3	3	3	3	3
mval-IF-3L-1C	ERC	4	4	3	4	4	4	3	4	4	4	3	4
mval-IF-3L-1C	PS	4	4	3	3	4	4	3	3	4	4	3	3
mval-IF-3L-1C	M	3	4	3	4	3	4	3	4	3	4	3	4
mval-IF-3L-2A	ERC	3	3	2	2	3	3	2	2	3	3	2	2
mval-IF-3L-2A	PS	2	2	2	2	2	2	2	2	2	2	2	2
mval-IF-3L-2A	M	2	2	2	2	2	2	2	2	2	2	2	2
mval-IF-3L-2B	ERC	3	2	2	2	3	2	2	2	3	2	2	2
mval-IF-3L-2B	PS	2	2	2	2	2	2	2	2	2	2	2	2
mval-IF-3L-2B	M	2	3	2	3	2	3	2	3	2	3	2	3
mval-IF-3L-2C	ERC	3	3	3	3	3	3	3	3	3	3	3	3

mval-IF-3L-2C	PS	3	3	3	3	3	3	3	3	3	3	3	3
mval-IF-3L-2C	M	3	4	3	3	3	4	3	3	3	4	3	3
mval-IF-3L-3A	ERC	2	2	2	2	2	2	2	2	2	2	2	2
mval-IF-3L-3A	PS	2	2	2	2	2	2	2	2	2	2	2	2
mval-IF-3L-3A	M	2	2	2	2	2	2	2	2	2	2	2	2
mval-IF-3L-3B	ERC	2	2	2	2	2	2	2	2	2	2	2	2
mval-IF-3L-3B	PS	2	3	2	2	2	3	2	2	2	3	2	2
mval-IF-3L-3B	M	2	3	2	3	2	3	2	3	2	3	2	3
mval-IF-3L-3C	ERC	2	2	2	2	2	2	2	2	2	2	2	2
mval-IF-3L-3C	PS	2	2	2	2	2	2	2	2	2	2	2	2
mval-IF-3L-3C	M	2	2	2	2	2	2	2	2	2	2	2	2
mval-IF-3L-4A	ERC	4	3	3	3	4	3	3	3	4	3	3	3
mval-IF-3L-4A	PS	3	3	3	3	3	3	3	3	3	3	3	3
mval-IF-3L-4A	M	3	3	3	3	3	3	3	3	3	3	3	3
mval-IF-3L-4B	ERC	4	4	3	4	4	4	3	4	4	4	3	4
mval-IF-3L-4B	PS	4	4	4	4	4	4	4	4	4	4	4	4
mval-IF-3L-4B	M	4	4	3	4	4	4	3	4	4	4	3	4
mval-IF-3L-4C	ERC	4	4	3	3	4	4	3	3	4	4	3	3
mval-IF-3L-4C	PS	4	4	4	4	4	4	4	4	4	4	4	4
mval-IF-3L-4C	M	4	4	4	4	4	4	4	4	4	4	4	4
mval-IF-3L-4D	ERC	4	4	4	4	4	4	4	4	4	4	4	4
mval-IF-3L-4D	PS	4	4	4	4	4	4	4	4	4	4	4	4
mval-IF-3L-4D	M	4	4	4	4	4	4	4	4	4	4	4	4
mval-IF-3L-5A	ERC	4	4	4	4	4	4	4	4	4	4	4	4
mval-IF-3L-5A	PS	4	4	4	4	4	4	4	4	4	4	4	4
mval-IF-3L-5A	M	5	4	4	4	5	4	4	4	5	4	4	4
mval-IF-3L-5B	ERC	4	4	4	4	4	4	4	4	4	4	4	4
mval-IF-3L-5B	PS	4	4	4	4	4	4	4	4	4	4	4	4
mval-IF-3L-5B	M	4	4	4	4	4	4	4	4	4	4	4	4
mval-IF-3L-5C	ERC	5	4	4	4	5	4	4	4	5	4	4	4
mval-IF-3L-5C	PS	4	4	4	4	4	4	4	4	4	4	4	4
mval-IF-3L-5C	M	5	5	4	5	5	5	4	5	5	5	4	5
mval-IF-3L-5D	ERC	4	5	4	4	4	5	4	4	4	5	4	4
mval-IF-3L-5D	PS	4	4	4	4	4	4	4	4	4	4	4	4
mval-IF-3L-5D	M	5	5	4	4	5	5	4	4	5	5	4	4
mval-IF-3L-6A	ERC	3	3	3	3	3	3	3	3	3	3	3	3
mval-IF-3L-6A	PS	3	3	3	3	3	3	3	3	3	3	3	3
mval-IF-3L-6A	M	3	3	3	3	3	3	3	3	3	3	3	3

mval-IF-3L-6B	ERC	3	3	3	3	3	3	3	3	3	3	3	3
mval-IF-3L-6B	PS	3	3	3	3	3	3	3	3	3	3	3	3
mval-IF-3L-6B	M	3	3	3	3	3	3	3	3	3	3	3	3
mval-IF-3L-6C	ERC	4	4	3	3	4	4	3	3	4	4	3	3
mval-IF-3L-6C	PS	4	4	3	4	4	4	3	4	4	4	3	4
mval-IF-3L-6C	M	3	4	3	3	3	4	3	3	3	4	3	3
mval-IF-3L-7A	ERC	4	4	4	4	4	4	4	4	4	4	4	4
mval-IF-3L-7A	PS	4	4	4	4	4	4	4	4	4	4	4	4
mval-IF-3L-7A	M	4	4	4	4	4	4	4	4	4	4	4	4
mval-IF-3L-7B	ERC	5	4	4	4	5	4	4	4	5	4	4	4
mval-IF-3L-7B	PS	4	4	4	4	4	4	4	4	4	4	4	4
mval-IF-3L-7B	M	4	5	4	4	4	5	4	4	4	5	4	4
mval-IF-3L-7C	ERC	5	5	5	5	5	5	5	5	5	5	5	5
mval-IF-3L-7C	PS	5	5	5	5	5	5	5	5	5	5	5	5
mval-IF-3L-7C	M	5	5	4	5	5	5	4	5	5	5	4	5
mval-IF-3L-8A	ERC	4	4	4	4	4	4	4	4	4	4	4	4
mval-IF-3L-8A	PS	3	4	3	4	3	4	3	4	3	4	3	4
mval-IF-3L-8A	M	4	4	4	4	4	4	4	4	4	4	4	4
mval-IF-3L-8B	ERC	4	4	3	3	4	4	3	3	4	4	3	3
mval-IF-3L-8B	PS	3	3	3	3	3	3	3	3	3	3	3	3
mval-IF-3L-8B	M	3	4	3	4	3	4	3	4	3	4	3	4
mval-IF-3L-8C	ERC	4	4	3	3	4	4	3	3	4	4	3	3
mval-IF-3L-8C	PS	4	4	4	4	4	4	4	4	4	4	4	4
mval-IF-3L-8C	M	4	4	3	3	4	4	3	3	4	4	3	3
mval-IF-3L-9A	ERC	5	5	4	4	5	5	4	4	5	5	4	4
mval-IF-3L-9A	PS	5	5	4	4	5	5	4	4	5	5	4	4
mval-IF-3L-9A	M	5	5	4	5	5	5	4	5	5	5	4	5
mval-IF-3L-9B	ERC	5	4	4	4	5	4	4	4	5	4	4	4
mval-IF-3L-9B	PS	4	4	4	4	4	4	4	4	4	4	4	4
mval-IF-3L-9B	M	5	5	4	4	5	5	4	4	5	5	4	4
mval-IF-3L-9C	ERC	5	5	4	5	5	5	4	5	5	5	4	5
mval-IF-3L-9C	PS	4	5	4	5	4	5	4	5	4	5	4	5
mval-IF-3L-9C	M	5	5	5	5	5	5	4	5	5	5	4	5
mval-IF-3L-9D	ERC	6	5	5	5	6	5	5	5	6	5	5	5
mval-IF-3L-9D	PS	5	6	5	5	5	6	5	5	5	6	5	5
mval-IF-3L-9D	M	5	6	5	5	5	6	5	5	5	6	5	5
mval-IF-3L-10A	ERC	5	5	5	5	5	5	5	5	5	5	5	5
mval-IF-3L-10A	PS	5	5	5	5	5	5	5	5	5	5	5	5

mval-IF-3L-10A	M	5	5	5	5	5	5	5	5	5	5	5	5
mval-IF-3L-10B	ERC	5	5	5	5	5	5	5	5	5	5	5	5
mval-IF-3L-10B	PS	5	5	5	5	5	5	5	5	5	5	5	5
mval-IF-3L-10B	M	5	5	5	5	5	5	5	5	5	5	5	5
mval-IF-3L-10C	ERC	5	5	5	5	5	5	5	5	5	5	5	5
mval-IF-3L-10C	PS	5	5	5	5	5	5	5	5	5	5	5	5
mval-IF-3L-10C	M	5	5	5	5	5	5	5	5	5	5	5	5
mval-IF-3L-10D	ERC	5	5	5	5	5	5	5	5	5	5	5	5
mval-IF-3L-10D	PS	5	5	5	5	5	5	5	5	5	5	5	5
mval-IF-3L-10D	M	5	5	5	5	5	5	5	5	5	5	5	5

A.2.2 Results on *lpr*, *mval* and *bccm-IF*

In this subsection results are shown for the deterministic and accelerated Neutral-Accelerated Tabu Search (NATS) setups when linked with the following constructive heuristics: *Efficient-Route-Cluster* (ERC); *Efficient-Route-Cluster-Random-Link* (ERCRL); *Path-Scanning* (PS); *Path-Scanning-Random-Link* (PSRL); *Improved-Merge* (IM); and *Randomised-Merge* (RM). The results are summarised in Table 7.3 in Chapter 7. Results are shown in Tables A.15 and A.16 for the setups, as well as the Memetic Algorithm of [6] on the MCARP *lpr* and *mval* set. Results are shown in Table A.17 for the same setups, as well as the Variable Neighbourhood Search (VNS) algorithm of [68] on the CARPTIF *bccm-IF* set.

Table A.15: Cost results for the Memetic Algorithm (MA) of [6] and for the multi-start and deterministic Neutral Accelerated Tabu Search (NATS) setups linked with different constructive heuristics on the *lpr* set when allowed 60 minutes of execution time.

		Multi-start			Deterministic		
Set	MA [6]	PSRL-NATS	RM-NATS	ERCRL-NATS	PS-NATS	IM-NATS	ERC-NATS
Lpr-a-01	13484	13484	13487	13484	13484	13501	13512
Lpr-a-02	28052	28069	28086	28052	28216	28150	28109
Lpr-a-03	76155	76201	76147	76278	76549	76708	76493
Lpr-a-04	127930	127753	127676	127919	127624	127719	127839
Lpr-a-05	206086	205581	205865	206068	204599	205494	206000
Lpr-b-01	14835	14835	14835	14835	14839	14835	14839
Lpr-b-02	28654	28654	28654	28654	28654	28654	28654
Lpr-b-03	77878	77907	78182	77934	77998	78484	78024
Lpr-b-04	127454	127538	127571	127551	127751	127246	127564
Lpr-b-05	212279	211881	212184	212285	210926	211054	211489
Lpr-c-01	18639	18639	18639	18639	18706	18666	18736
Lpr-c-02	36339	36339	36339	36361	36456	36408	36417
Lpr-c-03	111632	112029	111866	112198	111800	111725	112533
Lpr-c-04	169487	171033	169573	169534	171601	169749	169718
Lpr-c-05	260538	261428	260321	260780	259132	259883	260362

Table A.16: Cost results for the Memetic Algorithm (MA) of [6] and for the multi-start and deterministic Neutral Accelerated Tabu Search setups linked with different constructive heuristics on the *mval* set when allowed 60 minutes of execution time.

		Multi-start			Deterministic		
Set	MA [6]	PSRL-NATS	RM-NATS	ERCRL-NATS	PS-NATS	IM-NATS	ERC-NATS
mval-01A	230	230	230	230	233	233	230
mval-01B	261	261	261	261	294	261	267
mval-01C	315	315	316	316	369	316	331
mval-02A	324	324	324	324	324	324	324
mval-02B	395	395	395	395	401	395	395
mval-02C	526	534	545	544	551	565	569
mval-03A	115	115	115	115	118	116	118
mval-03B	142	142	142	142	142	146	144
mval-03C	166	166	166	168	166	170	172
mval-04A	580	580	580	580	590	580	594
mval-04B	650	650	650	650	654	652	652
mval-04C	631	630	633	632	630	649	653
mval-04D	776	756	763	769	762	795	798
mval-05A	597	599	597	599	610	608	614
mval-05B	615	615	617	619	619	627	623
mval-05C	697	697	701	697	697	709	701
mval-05D	757	739	741	741	739	754	751
mval-06A	326	326	326	326	335	329	336
mval-06B	317	317	320	317	327	323	342
mval-06C	375	371	371	376	378	375	392
mval-07A	364	364	364	364	377	364	370
mval-07B	412	414	412	412	415	421	416
mval-07C	428	428	428	432	435	440	437
mval-08A	581	581	581	585	581	589	603
mval-08B	531	531	536	531	532	543	553
mval-08C	638	638	643	644	635	656	667
mval-09A	458	458	458	458	459	465	458
mval-09B	453	453	459	453	456	465	461
mval-09C	434	434	436	434	434	435	435
mval-09D	520	520	520	520	520	534	532
mval-10A	634	643	643	643	643	654	647
mval-10B	662	662	662	662	663	669	666
mval-10C	624	624	624	624	624	628	635
mval-10D	650	650	650	650	650	672	651

Table A.17: Cost results for the Variable Neighbourhood Search (VNS) algorithm of [68] and for the multi-start and deterministic Neutral Accelerated Tabu Search setups linked with different constructive heuristics on the *bccm-IF* set when allowed 60 minutes of execution time.

		Multi-start			Deterministic		
Set	VNS [68]	PSRL-NATS	RM-NATS	ERCRL-NATS	PS-NATS	IM-NATS	ERC-NATS
val1A-IF	229	229	229	229	251	253	232
val1B-IF	229	229	238	229	251	253	232
val1C-IF	259	270	271	269	275	291	296
val2A-IF	434	480	434	480	487	532	557
val2B-IF	434	480	434	480	487	532	557
val2C-IF	488	532	532	599	553	590	597
val3A-IF	120	120	120	122	123	125	123
val3B-IF	120	120	120	120	123	125	123
val3C-IF	126	141	132	140	146	151	159
val4A-IF	533	538	538	538	587	568	583
val4B-IF	533	538	538	538	587	568	583
val4C-IF	533	538	538	568	568	568	583
val4D-IF	538	574	559	578	604	602	604
val5A-IF	879	925	925	1084	929	929	1239
val5B-IF	879	925	925	1013	929	929	1239
val5C-IF	879	925	925	1019	929	929	1239
val5D-IF	879	925	925	1089	929	931	1239
val6A-IF	343	343	343	343	344	344	344
val6B-IF	343	343	343	343	344	344	344
val6C-IF	367	371	373	375	383	402	371
val7A-IF	444	444	444	444	444	447	444
val7B-IF	444	444	444	444	444	447	444
val7C-IF	444	444	444	444	453	447	444
val8A-IF	545	548	551	551	551	591	603
val8B-IF	545	548	554	549	551	591	603
val8C-IF	547	568	575	563	589	579	630
val9A-IF	-	569	569	565	598	572	627
val9B-IF	546	564	569	570	598	572	627
val9C-IF	-	570	567	574	598	572	627
val9D-IF	546	567	565	570	570	595	625
val10A-IF	-	672	641	676	687	680	683
val10B-IF	-	674	676	677	687	680	683
val10C-IF	-	674	647	671	687	680	683
val10D-IF	-	674	674	678	680	678	679