Rule-based design of a granular tabu search for the multi-depot vehicle routing problem

Working Paper DPO-2021-01 (version 1, 21.05.2021)

Christian Becker **Rossana Cavagnini** **Stefan Irnich** **Michael Schneider**

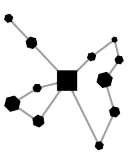
{schroeder|cavagnini|schneider}@dpo.rwth-aachen.de
Deutsche Post Chair – Optimization of Distribution Networks
RWTH Aachen University, Germany

irnich@uni-mainz.de
Chair of Logistics Management, Gutenberg School of Management and Economics
Johannes Gutenberg University, Mainz, Germany

Abstract

We describe a rule-based process to design a granular tabu search for the multi-depot vehicle routing problem. Using this process, only well-known and mature algorithmic components from the literature are required to achieve a reasonable performance compared to state-of-the-art metaheuristics. In each step, we analyze a single algorithmic component by comparing variants resulting from one or more alternative designs of this component against a fixed base variant. The selection of the best design for each component relies on two clear rules that are based on the Wilcoxon signed-rank test and a straightforward tradeoff measure. The decisions on the algorithmic components are partly in line with previous recommendations for other VRP variants from the literature but also contradict in some important aspects.

Keywords: *vehicle routing, multi-depot, granular tabu search, rule-based design*



Deutsche Post
Chair - Optimization of
Distribution Networks

RWTH AACHEN
UNIVERSITY

1 Introduction

This paper uses a rule-based process to design a *granular tabu search* (GTS, Toth and Vigo 2003) for the *multi-depot vehicle routing problem* (MDVRP, Cordeau et al. 1997). The classical MDVRP extends the *capacitated VRP* (CVRP, Toth and Vigo 2002, Vigo and Toth 2014) by considering more than one depot from which a limited fleet of vehicles performs routes, each starting and ending at the same depot. Applications arise in the distribution of, e.g., food (Cassidy and Bennett 1972, Pooley 1994), chemical and industrial products (Ball et al. 1983, Bell et al. 1983, Brown et al. 1987), and machines (Fisher et al. 1982).

The TS paradigm (Glover 1989) embeds a local search that iteratively modifies a solution by means of elementary neighborhood operators. It allows to escape from local optima by applying the best non-tabu move in each iteration, even if it is deteriorating, and uses a memory to avoid cycling. In GTS, originally proposed by Toth and Vigo (2003), the neighborhoods are explored by looping over a so-called list of *generator arcs*. A generator arc (i, j) in combination with a neighborhood operator uniquely identifies a move, and after the application of the move, the arc (i, j) is contained in the resulting solution. To accelerate the neighborhood exploration, the generator arc list is restricted to promising generator arcs. Although this biases the exploration towards promising arcs, also less promising arcs can be inserted because all neighborhood operators insert more than one arc. The entire process of selecting and sorting the arcs of the generator arc list is called *granularization*, and the resulting reduced neighborhoods are called *granular* neighborhoods.

Comparisons (e.g., found in Vidal et al. 2012) show that GTS implementations stand out because of their speed, i.e., they provide high-quality solutions within strongly reduced runtimes compared to other metaheuristics for VRP variants: examples are the GTS of Toth and Vigo (2003) for the CVRP, of Schneider et al. (2017) for the *VRP with time windows* (VRPTW), or of Escobar et al. (2014b) for the classical MDVRP. The latter work defines the set of generator arcs to include all feasible arcs with a cost lower than a granularity threshold value and all arcs belonging to the best feasible solution. Although their GTS cannot match the solution quality of the metaheuristic of Vidal et al. (2012) in computational experiments on the instances of Cordeau et al. (1997), it remains until now the fastest competitive heuristic for solving the classical MDVRP on this standard benchmark set.

We identify the following gaps in the research on GTS for the MDVRP: First, the GTS of Escobar et al. (2014b) is the fastest known algorithm for the MDVRP, but it has only been evaluated on the instances of Cordeau et al. (1997), and comparisons on instances with different characteristics are missing. Second, no studies have systematically investigated the impact of the design of the GTS components on the algorithmic performance in the multi-depot case. The recommendations provided by Toth and Vigo (2003) for the CVRP and by Schneider et al. (2017) for the VRPTW cannot directly be translated to the multi-depot context.

The contribution of this paper is to describe the design of a GTS which is applied to different MDVRP instance sets and shows a reasonable performance compared to the state-of-the-art metaheuristics for the MDVRP. We show that only well-known and mature algorithmic components from the literature are required to achieve this if the design follows a rule-based process to decide between different alternatives for the components of the GTS. In each step of the process, we analyze a single algorithmic component by comparing variants resulting from one or more alternative designs of this component against a fixed base variant. The selection of the best design for each component relies on two clear rules that are based on the *Wilcoxon signed-rank test* (WSRT, Wilcoxon 1945) and a straightforward tradeoff measure to decide between designs

in case several of them lead to significant improvements with regard to the WSRT. The advantage of this process, apart from its statistical validity, is its reproducibility.

We investigate two types of algorithmic components in computational experiments: First, we evaluate the impact of classical design choices regarding the neighborhood search in the GTS (the set of neighborhood operators, the order in which they are explored, and the pivoting rule to use). Second, we systematically investigate the granularity features (the sparsification method and strength, the additional arcs to be inserted into the generator arc list, and the use of dynamic sparsification).

As an outcome of the rule-based selection of the designs, we propose two final GTS configurations, one focusing on speed and another one on quality. Both show decent performance on all available benchmark sets. Moreover, the analyses of the experiments help to better understand which of the recommendations for single-depot problems given by Toth and Vigo (2003) and Schneider et al. (2017) also apply to the MDVRP, and which general recommendations on designing granular local searches (Schröder et al. 2020) transfer to our GTS for the MDVRP.

The remainder of this paper is organized as follows. Section 2 reviews the pertinent literature, and Section 3 provides a detailed description of the components of the GTS framework that we use. Section 4 details the study design and presents the computational results. Because the study in Section 4 analyzes the impact of the GTS components introduced in Section 3, there is unavoidably some redundancy in the two sections. To keep the redundancy as small as possible, we focus on a general description in Section 3, while we provide concrete parameter settings for the design of components in Section 4. The paper closes with a discussion of our results and suggests future research directions in Section 5.

2 Literature review

The MDVRP belongs to the class of NP-hard problems because it includes the CVRP as a special case. Due to its computational complexity and practical relevance, the research community has dedicated a lot of effort to develop both exact and heuristic solution approaches for the MDVRP. The most successful exact methods are based on set-partitioning formulations strengthened by different classes of valid inequalities and solved by branch-price-and-cut approaches (Baldacci and Mingozzi 2009, Contardo and Martinelli 2014, Sadykov et al. 2021). The current limit for consistently solving MDVRP instances to optimality lies at about 300 customers.

To address larger instances (or achieve shorter runtimes), a multitude of modern heuristics based on different paradigms have been proposed: adaptive large neighborhood search (Pisinger and Ropke 2007), genetic algorithms (Thangiah and Salhi 2001, Ombuki-Berman and Hanshar 2009, Lau et al. 2009, Vidal et al. 2012, 2014), *iterated local search* (ILS, Gauthier and Irnich 2020), matheuristics (Subramanian et al. 2013, Accorsi and Vigo 2020), and TS (Renaud et al. 1996, Cordeau et al. 1997, 2001, Cordeau and Maischberger 2012, Escobar et al. 2014b). Among these modern heuristics, the current state-of-the-art implementation for the MDVRP is the *hybrid genetic search with advanced diversity control* (HGSADC) of Vidal et al. (2012), later extended to HGSADC+ by introducing an efficient dynamic programming component that assigns customers to depots, determines the first visited customer within each route, and provides route choices (Vidal et al. 2014). The quality of HGSADC+ can be attributed to the use of a memory which keeps track of good solution features and a mechanism to guarantee population diversity, which achieves a good level of diversification in the solution process.

The main focus of the paper at hand lies on the design of granular neighborhoods. Since their introduction by Toth and Vigo (2003) for the CVRP, numerous papers have successfully used this approach within different metaheuristics—TS (Prins et al. 2007, Kirchler and Wolfler Calvo 2013), variable neighborhood search (Escobar et al. 2014a), or memetic algorithm (Vidal et al. 2012)—to address different routing problem like variants of the VRP (Toth and Vigo 2003, Escobar et al. 2014b, Schneider et al. 2017), the team orienteering problem (Labadie et al. 2012), the dial-a-ride problem (Kirchler and Wolfler Calvo 2013), or the capacitated location routing problem (Escobar et al. 2014a, Schneider and Löffler 2019).

Several papers have given sporadic recommendations on the design of granular searches (e.g., Toth and Vigo 2003), and Schneider et al. (2017) and Schröder et al. (2020) have even carried out systematic investigations to give recommendations. Schneider et al. (2017) develop a GTS for the VRPTW and investigate different alternatives of realizing granular neighborhoods, i.e., different sparsification methods, sparsification strengths, and mechanisms for including additional arcs into the list of generator arcs. Instead, Schröder et al. (2020) study a pure granular local search for the CVRP and put the main focus on the design decisions in local search, i.e., the set of utilized neighborhood operators, the order in which they are explored, the pivoting rule etc. With regard to granular search, only the effect of different sparsification strengths is investigated.

3 Granular tabu search for multi-depot VRPs

This section describes the general GTS framework and details the algorithmic components. We start by formalizing (i) the MDVRP and (ii) the neighborhood search. The MDVRP is defined on a complete directed graph $\mathscr{G} = (\mathscr{V}, \mathscr{A})$ with vertex set $\mathscr{V}$ and arc set $\mathscr{A}$. The vertex set $\mathscr{V}$ consists of *m depot vertices* $\mathscr{V}_d = \{1,\dots,m\}$ and *n customer vertices* $\mathscr{V}_c = \{m+1,\dots,m+n\}$. The arc set $\mathscr{A}$ is the union of the *customer arcs* $\mathscr{A}_c = \{(i,j) \mid i,j \in \mathscr{V}_c, i \neq j\}$ and *depot arcs* $\mathscr{A}_d = \{(i,j) \mid i \in \mathscr{V}_c, j \in \mathscr{V}_d \vee i \in \mathscr{V}_d, j \in \mathscr{V}_c\}$. A non-negative *cost* c_{ij} and a *travel time* t_{ij} are associated with each arc $(i,j) \in \mathscr{A}$. Moreover, a non-negative *demand* q_i and *service time* s_i are associated with each vertex $i \in \mathscr{V}$ assuming $q_i = s_i = 0$ for the depots $i \in \mathscr{V}_d$. A *route* R is a cycle in $\mathscr{G}$ that includes exactly one depot vertex. We use the standard notation $\mathscr{V}(R)$ and $\mathscr{A}(R)$ to refer to vertices and arcs of R, and $w(S)$ as shorthand notation for $\sum_{s\in S} w_s$ for any set S and coefficients $(w_s)_{s\in S}$ defined for $s \in S$. A route is *feasible* with respect to the vehicle capacity Q and maximum duration T if $q(\mathscr{V}(R)) \leq Q$ and $s(\mathscr{V}(R)) + t(\mathscr{A}(R)) \leq T$, respectively. The *cost* of a route is $c(\mathscr{A}(R))$. Each depot hosts a homogeneous fleet of k vehicles that must return to their starting depots. A solution x of the MDVRP is a set of routes. We denote by X the set of all solutions. A solution $x \in X$ is *feasible* if all its routes are feasible, and the routes include each customer vertex exactly once and each depot not more than k times. A feasible solution is *optimal* if it minimizes the sum of the costs of its routes.

A neighborhood $\mathscr{N}$ is a map $\mathscr{N} : X \to 2^X$, i.e., for a given solution $x \in X$, the neighborhood consists of a set of solutions $\mathscr{N}(x) \subset X$ (not necessarily feasible). Every $x' \in \mathscr{N}(x)$ is called *neighbor* of x. A neighbor $x' \in \mathscr{N}(x)$ is *improving* with respect to a cost function c if $c(x') < c(x)$ (we only consider minimization). Local search starts from a given solution, iteratively selects an improving neighbor (as long as one exists), and replaces the current solution by the neighbor. The process of identifying an improving (or a best non-improving) neighbor is called *neighborhood exploration*, and the termination criterion for the neighborhood exploration is often referred to as *pivoting rule* (e.g., first improvement or best improvement). A solution without improving neighbor is a local optimum. We explore neighborhoods using the generator arc principle (Toth and Vigo 2003), and we denote the generator arc list as $\mathscr{A}_g$.

Algorithm 1 gives an overview of the GTS algorithm that we detail in the following sections. We allow infeasible solutions in the construction heuristic and in the improvement phase of the GTS. The degree of infeasibility is controlled with a generalized cost function C discussed in Section 3.1. In Section 3.2, we briefly describe the construction heuristic (Line 1).

Algorithm 1: Pseudocode of GTS.

```
1 x ← construction_heuristic()
2 initialize(tabu_list, generalized cost function C with penalty factors α and β, generator arc list A_g)
3 repeat
4 |   x° ← neighborhood_exploration(x, N, C, tabu_list, aspiration, A_g)
5 |   update(tabu_list, (α, β), A_g)
6 |   x ← x°
7 until stopping criterion met
```

Lines 2–6 describe the initialization and iteration of the GTS. The *tabu list* stores the arcs that have been removed in previous iterations. The tabu criterion prohibits the re-insertion of those arcs for a number of iterations denoted as the *tabu tenure* τ. The tabu tenure τ is a random number drawn from the interval $[\tau_{min}, \tau_{max}]$. The aspiration criterion allows moves that are tabu if they lead to a new best solution.

In each iteration, the neighborhood exploration (Line 4) depends on the used neighborhood operators (Section 3.3), the exploration strategy (Section 3.4), and the granularization (Section 3.5). Next, the tabu list, the penalties, the generator arc list $\mathscr{A}_g$ (Line 5), and the current solution (Line 6) are updated. We use gain-based pruning to further speed up the search (Section 3.6). A continuous diversification mechanism ensures the exploration of new areas of the solution space (Section 3.7). GTS stops when a time limit t_{max} or a maximum number of iterations without improvement η_{max} is reached.

3.1 Generalized cost function

Because we cannot guarantee to find a feasible solution with the construction heuristic, we need to handle infeasible solutions. Furthermore, allowing infeasible solutions in local search-based heuristics has been shown to be vital for their performance and is therefore widely used in the literature (see, e.g., Cordeau et al. 1997). We allow infeasible solutions with respect to both capacity and tour duration constraints. We use a generalized cost function (see, e.g., Gendreau et al. 1994) that adds normalized penalties for capacity and duration violations to the routing costs $c(x)$:

$$C(x) = c(x) + norm_\alpha \cdot \alpha \cdot viol_\alpha(x) + norm_\beta \cdot \beta \cdot viol_\beta(x). \tag{1}$$

Both penalties are a product of a penalty normalization coefficient ($norm_\alpha, norm_\beta$), a penalty factor (α, β), and the total violation of the respective constraint ($viol_\alpha, viol_\beta$) determined as $viol_\alpha(x) = \sum_{R \in x} \max\{0, q(\mathscr{V}(R)) - Q\}$ and $viol_\beta(x) = \sum_{R \in x} \max\{0, s(\mathscr{V}(R)) + t(\mathscr{A}(R)) - T\}$.

The penalty normalization coefficients ($norm_\alpha, norm_\beta$) have been first suggested by Schneider and Löffler (2019) and are calculated as follows:

$$norm_\alpha = \frac{n \cdot c(x^\circ)}{\sum_{i \in \mathcal{V}} q_i}, \tag{2a}$$

$$norm_\beta = \frac{n \cdot c(x^\circ)}{\text{total duration}} = \frac{n \cdot c(x^\circ)}{\sum_{i \in \mathcal{V}} s_i + t(x^\circ)}. \tag{2b}$$

Normalization coefficients make GTS more adaptive to instances that strongly differ in the spatial distribution of customers and depots, demand ratios, or service and travel times.

The penalty factors $\rho \in \{\alpha, \beta\}$ are initialized to a value of ρ_{init}. In each GTS iteration, the penalty factors are updated as follows: if the new solution x° is feasible with respect to capacity or/and duration, the respective penalty factor is divided by $\Delta_{pen} > 1$. Otherwise, the penalty factor is multiplied by Δ_{pen}. As a result, several consecutive infeasible solutions increase penalties and thus guide the search towards feasible solutions. Conversely, a number of consecutive feasible solutions make infeasible solutions more attractive.

To stabilize the penalization process, penalty factors are controlled such that after finding a feasible (an infeasible) solution the penalty factor is not overly large (small). The following update formulas uses the interval $[\rho_{min}, \rho_{max}]$ to achieve this:

$$\Delta_{pen} = \exp\left(\frac{\log\left(\frac{\rho_{max}}{\rho_{min}}\right)}{\eta_{pen}}\right), \tag{3a}$$

$$\rho = \begin{cases} \min\left(\rho_{max}, \frac{\rho}{\Delta_{pen}}\right), & \text{if } x^\circ \text{ is feasible} \\ \max\left(\rho_{min}, \rho \cdot \Delta_{pen}\right), & \text{if } x^\circ \text{ is infeasible.} \end{cases} \tag{3b}$$

Δ_{pen} is defined in such a way that after a maximum of η_{pen} consecutive improvements (deteriorations) the penalty factor ρ reaches ρ_{min} (ρ_{max}). Pretests have shown that this penalty update rule is superior to the original rule suggested in Toth and Vigo (2003).

3.2 Construction heuristic

To construct an initial solution, we use an insertion heuristic that tries to generate a feasible solution but is also able to work with infeasible solutions in case a feasible solution cannot be found. Infeasible solutions are assessed using the generalized cost function C defined in (1). The insertion heuristic starts with an empty solution x and opens a first route by connecting a depot with a customer in the cheapest way. In each iteration, the two options of either opening a new route or inserting an unassigned customer into an existing route are compared, and the option with the smallest generalized cost is carried out. To obtain a feasible starting solution, we strongly penalize violations with high penalty factors α and β of 100 000.

To accelerate the construction heuristic, we reduce the number of evaluated customer insertions in each iteration. We keep two lists, one containing the yet unassigned customers and the other containing the already inserted ones. We calculate a so-called *pool size* n_{pool} by first drawing a random number from the interval $[cnstr_l, cnstr_u]$ and multiplying it with the number n of customers. In each insertion step, we shuffle the two lists and only evaluate the insertion of the first n_{pool} customers in the list of unassigned customers before the first n_{pool} customers in the list of already inserted customers.

3.3 Neighborhoods

For our *base variant* of the GTS (in the following denoted as `base`), we use the four neighborhoods used by Toth and Vigo (2003), namely relocate, exchange, 2-opt, and 2-opt*. 2-opt is only used as intra-route neighborhood and 2-opt* as inter-route neighborhood. We give a very brief description of these *classical* neighborhoods (for more details, see, e.g., Funke et al. 2005).

relocate (rel) moves a customer vertex from its current position to a different position in the same or a different route.

exchange (ex) swaps two customer vertices within the same or between different routes.

2-opt (2) removes two non-consecutive arcs from a route, inverts the vertex segment between the removed arcs, and reconnects the inverted segment using two new arcs.

2-opt* (2*) removes one arc from each route and mutually reconnects the first part of one route with the second part of the other route.

In our studies, we also consider GTS variants that additionally use subsets of the following four *string* neighborhoods, which we implement such that overlaps concerning the moves generated by the classical operators are avoided.

string-relocate (strel) relocates a string of ℓ consecutive customer vertices within the same route or between different routes. The string length is restricted to $2 \leq \ell \leq \ell_{max}$ for a given maximum length ℓ_{max}.

string-relocate-inverted (streli) same as strel but the string is inverted before it is reinserted.

string-exchange (stex) exchanges two non-overlapping customer vertex strings of up to ℓ_{max} vertices either from the same or from different routes. We require both strings to comprise at least three vertices together.

string-exchange-inverted (stexi) same as stex but both strings are inverted before they are reinserted.

To limit the search effort, we set the maximum string length to $\ell_{max} = 3$. As a result, all considered neighborhoods are quadratic in the number of customers.

Figure 1 illustrates the neighborhood moves. A plus sign indicates the direct successor and a minus sign the direct predecessor of a vertex in the current solution x. Similarly, $\ell+$ indicates the ℓth successor and $\ell-$ the ℓth predecessor of a vertex.

Recall that for the neighborhoods $o \in \{\text{2-opt, 2-opt*, relocate, exchange}\}$, the arc (i, j) uniquely determines the neighbor $x' \in \mathcal{N}(x)$, cf. Figures 1a–d. We can therefore precisely describe a *neighborhood move* as a function $move : X \to X$ defined as $x' = move((i, j), o, x)$. For the string neighborhoods $o \in \{\text{strel, streli, stex, stexi}\}$, a constant number of cases for the string length(s) must be considered to uniquely identify a move, cf. Figures 1e–h. For the sake of simplicity, we slightly abuse notation and write $x' = move((i, j), o, x)$ instead of $move((i, j), \ell, o, x)$ or $move((i, j), \ell_1, \ell_2, o, x)$ for these neighborhoods.

Relocate, 2-opt*, string-relocate, and string-relocate-inverted are able to close routes; however, the classical and string operators lack the ability to open new routes. Because we temporarily allow infeasible solutions during the search and we address a multi-depot routing problem, this limitation is likely to negatively influence the performance of the GTS. For example, opening a new route could lead to substantial improvements by reducing constraint violations, but it is not possible. Therefore, we introduce four so-called *split operators*, which split one route into two routes, and we investigate their effect on the performance of GTS:

split-one removes one customer from its route and forms a new route with only this customer.

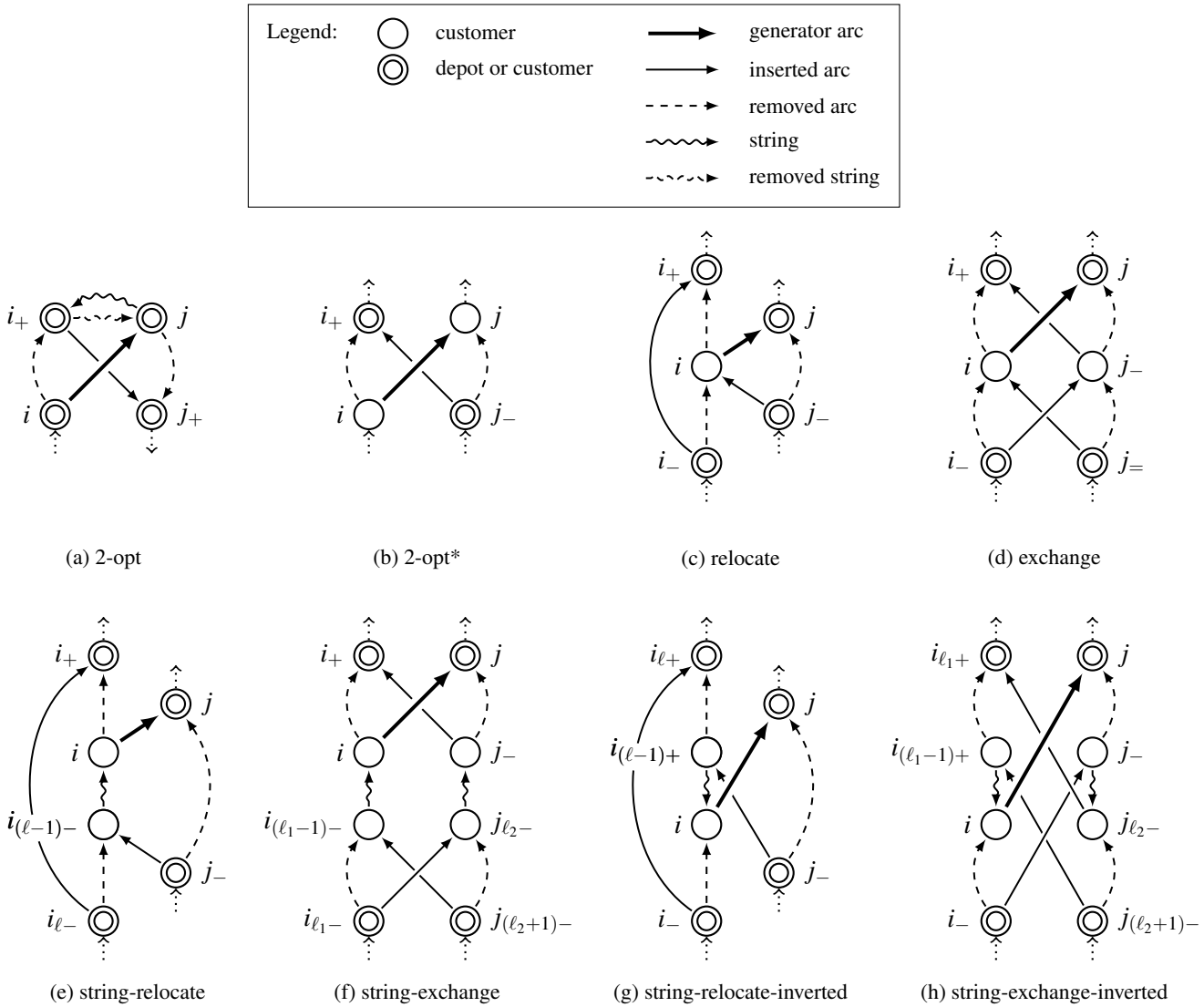


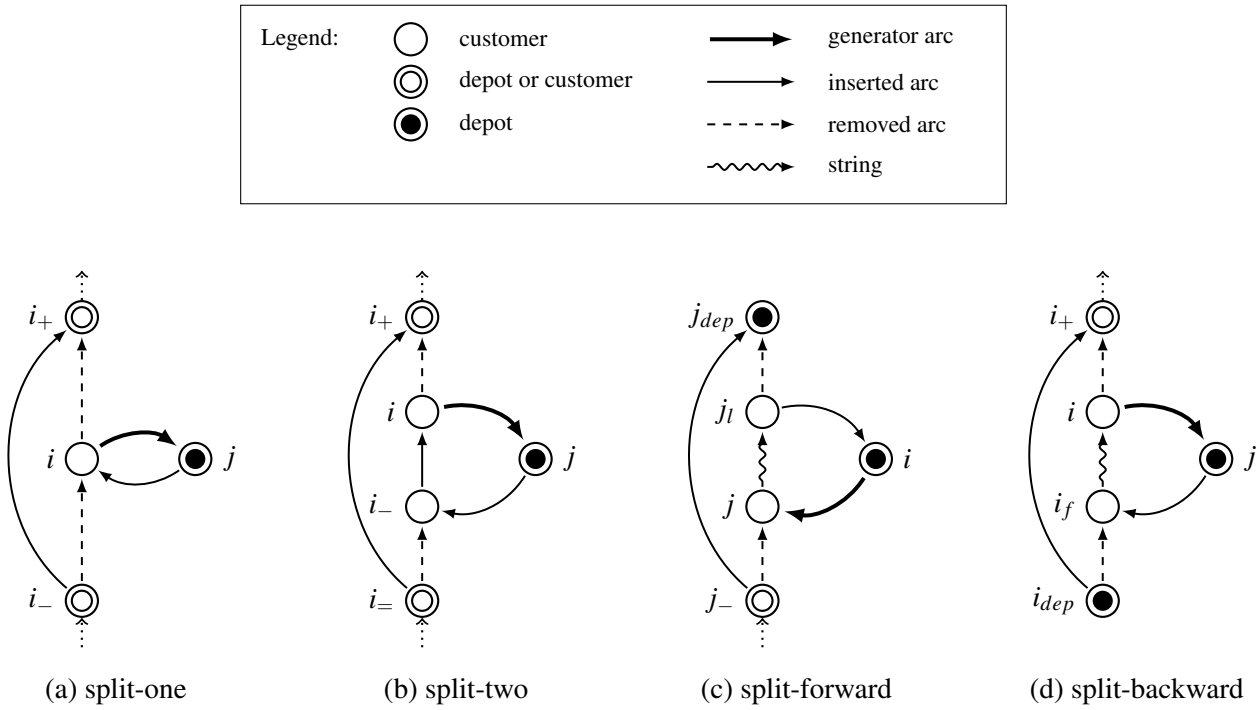
Figure 1: Classical (a–d) and string (e–h) neighborhood operators of GTS

split-two removes two consecutive customers from their route and forms a new route with only these two customers.

split-forward removes the last part of a route and forms a new route with it.

split-backward removes the first part of a route and forms a new route with it.

Figure 2 illustrates the split operators. Here, a vertex indexed with l represents the last vertex of a route before the depot, a vertex indexed with f represents the first vertex of a route after the depot.



(a) split-one (b) split-two (c) split-forward (d) split-backward

Figure 2: Split neighborhood operators of GTS

In the next sections, when describing different variants, we denote the selected neighborhood operators for the respective variant by O.

3.4 Exploration strategy

In the original GTS of Toth and Vigo (2003), the classical neighborhoods were combined into a single composite neighborhood and explored with a best-improvement pivoting rule. In this section, we describe alternatives to the original design. Under the term *exploration strategy*, we summarize the decisions on (i) the order in which neighbors of the different (classical, string, and split) neighborhoods are explored (Line 4 of Algorithm 1) and (ii) whether and when the exploration is terminated before all neighbors are inspected.

Composite neighborhood: For variant `base`, we use a composite neighborhood defined by the union of all neighborhoods, i.e., $\mathcal{N}(x,\mathcal{A}_g) = \bigcup_{o\in O}\mathcal{N}_o(x,\mathcal{A}_g)$. With best improvement, the order in which operators of O are considered is irrelevant.

To achieve a balanced evaluation also for the first-improvement rule, we implement the exploration algorithm by traversing the generator arc list in an outer loop (see Algorithm 2). Then, for each generator arc $(i,j) \in \mathcal{A}_g$, the inner loop considers each neighborhood operator $o \in O$ (in the order given in Section 3.3) and evaluates the move $move((i,j),o,x)$.

VND-like: Another popular scheme is *variable neighborhood descent* (VND), which explores two or more neighborhoods in a fixed order. (We have chosen the term *VND-like* to stress that this strategy is used for a single exploration, so that the resulting neighbor x° is not necessarily a local minimum with respect to all neighborhoods.) We implement an outer loop that first traverses the neighborhood operators $o \in O$ in the order given in Section 3.3, see Algorithm 3. The inner loop then considers the generator arcs (i,j) and thus moves $move((i,j),o,x)$ in the order given by the list $\mathscr{A}_g$. If an improving move is found in the current neighborhood, no further neighborhoods are evaluated. Compared to the composite neighborhood, we note the swapped outer and inner loops and the additional termination condition in Line 16.

Algorithm 2: Neighborhood exploration with composite neighborhood

```
1  Input: current solution x
2         neighborhoods 𝒩 def. by operators O
3         generalized cost function C
4         tabu_list, aspiration
5         sparse generator arc list 𝒜_g
6  x° ← ∅; C(x°) ← ∞
7  for (i,j) ∈ 𝒜_g
8      for o ∈ O
9          x' ← move((i,j),o,x)
10         if x' ∈ tabu_list and not aspiration
11             continue
12         if C(x') < C(x°)
13             x° ← x'
14             if first impr. and C(x°) < C(x)
15                 return x°
16 return x°
```

Algorithm 3: Neighborhood exploration with VND-like strategy

```
1  Input: current solution x
2         neighborhoods 𝒩 def. by operators o ∈ O
3         generalized cost function C
4         tabu_list, aspiration
5         sparse generator arc list 𝒜_g
6  x° ← ∅; C(x°) ← ∞
7  for o ∈ O
8      for (i,j) ∈ 𝒜_g
9          x' ← move((i,j),o,x)
10         if C(x') < C(x°)
11             x° ← x'
12             if x' ∈ tabu_list and not aspiration
13                 continue
14             if first impr. and C(x°) < C(x)
15                 return x°
16     if C(x°) < C(x)
17         return x°
18 return x°
```

Besides the selection of arcs to include into the generator arc list, the order of the generator arcs can have a critical influence on the performance of the GTS. When using the first-improvement pivoting rule, neighborhood exploration might be terminated before the complete generator arc list has been evaluated. A small effect may also result for best improvement when combined with gain-based pruning (see Section 3.6) because good moves should be found early when exploring a neighborhood. We quantify the effect of different sortings in the numerical studies.

3.5 Granular neighborhoods

Constructing a good generator arc list is non-trivial. Roughly speaking, the generator arc list is determined by a sparsification method, which first sorts all arcs $(i,j) \in \mathscr{A}$ according to a criterion, e.g., increasing costs c_{ij}, and a sparsification factor π, which specifies the share of the arcs which are then inserted into the generator arc list $\mathscr{A}_g$, i.e., the first $\lceil \pi \cdot |\mathscr{A}| \rceil$ arcs according to the sorting.

Recall that a generator arc $(i,j) \in \mathscr{A}_g$ in combination with a neighborhood operator $o \in O$ (and the length(s) of the string(s) in case of the string neighborhoods) uniquely identifies a move, i.e., a neighbor solution is determined via $x' = move((i,j),x,o)$. Neighborhoods are explored by looping over the list of generator arcs in each iteration of the GTS. Reducing the generator arc list can accelerate the neighborhood exploration and speed up the search. Therefore, the main task of granularization is the identification of the most promising

arcs to be inserted into the generator arc list $\mathscr{A}_g$ and to keep that list as short as possible without compromising solution quality (too much).

The design of the granularization comprises several decisions, which are discussed in the following subsections:

- The *sparsification method* sorts arcs according to how promising they are as generator arcs (Section 3.5.1).
- The *sparsification level* decides whether the sparsification method is applied to the arcs of the whole graph or whether the sparsification is done on a per-customer basis (Section 3.5.2).
- The *sparsification factor* determines the size of $\mathscr{A}_g$ and thus the sparsification strength. *Dynamic sparsification* dynamically adapts the sparsification strength to balance between intensification and diversification (Section 3.5.3).
- Including *additional arcs* into $\mathscr{A}_g$ like, e.g., arcs that have previously been inserted into accepted solutions, is known to improve the quality of the granularization (Section 3.5.4).
- Techniques to *select the depot arcs* to be included in $\mathscr{A}_g$ are also important for the quality of the granularization (Section 3.5.5).

Finally note that in all split neighborhoods, the generator arcs are depot arcs, i.e., $(i,j) \in \mathscr{A}_d$ (see Figure 2). Because the number of depot arcs is relatively small ($2 \cdot n \cdot |\mathscr{V}_d|$) and using all of them has a strong positive effect on solution quality (see Schneider et al. 2017), we treat split neighborhoods differently than the classical and string neighborhoods. The generator arc list used for a split neighborhood operator contains all depot arcs $\mathscr{A}_d$, so that split neighborhoods are not granular.

3.5.1 Sparsification method

In the literature, original costs c_{ij} are the most popular criterion to sort the arcs $(i,j) \in \mathscr{A}$, but reduced costs computed from a network relaxation have also been successfully used (Labadie et al. 2012, Schneider et al. 2017). To compute reduced costs for the MDVRP, we use the following MDVRP relaxation that does not consider capacity, tour duration, and subtour elimination constraints:

$$\min \sum_{(i,j)\in\mathscr{A}} c_{ij}x_{ij} \tag{4a}$$

$$\text{subject to} \sum_{h\in\mathscr{V}_c} x_{hi} = \sum_{j\in\mathscr{V}_c} x_{ij} = 1 \qquad \forall i \in \mathscr{V}_c \tag{4b}$$

$$\sum_{i\in\mathscr{V}_c} x_{id} = \sum_{j\in\mathscr{V}_c} x_{dj} = k_d \qquad \forall d \in \mathscr{V}_d \tag{4c}$$

$$\sum_{d\in\mathscr{V}_d} k_d = k_{all} \tag{4d}$$

$$\left\lceil \left(\sum_{i\in\mathscr{V}_c} q_i \right) / Q \right\rceil \leq k_{all} \leq k \cdot m \tag{4e}$$

$$x_{ij} \geq 0 \qquad \forall (i,j) \in \mathscr{A} \tag{4f}$$

$$k_{all}, k_d \geq 0 \qquad \forall d \in \mathscr{V}_d. \tag{4g}$$

In this model, k_{all} is the total number of used vehicles, and k_d is the number of used vehicles hosted at depot d. This linear program can be solved by any LP solver. Variant `base` uses original costs c_{ij}, and the alternative variant sorts the arcs (i,j) in order of increasing reduced costs of the respective x_{ij}-variables of a solution of model (4).

3.5.2 Sparsification level

The sparsification can be performed either on the whole graph or on a per-customer basis. In the first case, all arcs $(i, j) \in \mathscr{A}$ are sorted using the sparsification method, and the generator arc list $\mathscr{A}_g$ contains the initial subsequence of the sorted list ending at a predefined threshold. Sparsification on the whole graph level can be detrimental for VRP instances that contain isolated customers because it can result in generator arc lists $\mathscr{A}_g$ that contain no or only very few arcs incident to these isolated customers.

To counteract this effect, per-customer sparsification first sorts the incident arcs of each customer. Then, for each customer, a subsequence (starting from the first arc up to a predefined threshold) is selected, and the subsequences are merged into the list of generator arcs $\mathscr{A}_g$. The generator arc list $\mathscr{A}_g$ is sorted according to the criterion used in the sparsification method.

In our studies, we compare three variants: First, variant `base` sparsifies on the whole graph level. Second, connections from isolated customers are inserted after sparsification on the whole graph level (see Schneider et al. 2017). A third variant directly sparsifies on a per-customer basis.

3.5.3 Dynamic sparsification

We use a sparsification factor π to control the size of the generator arc list $\mathscr{A}_g$. For example, a sparsification factor of 0.2 selects the first 20% of the sorted arc list. On the per-customer basis, the first 20% most promising arcs of each customer are selected first, then merged into the generator arc list and sorted. Dynamic sparsification, i.e., adapting the strength of the sparsification during the search, has been successfully used in several applications of granular search (e.g., Labadie et al. 2012, Escobar et al. 2014b).

In our studies, we compare two alternative variants of dynamic sparsification. Variant `base` uses the well-known dynamic sparsification mechanism of Toth and Vigo (2003): It consists of two alternating phases, an intensification phase and a diversification phase. The method starts with an intensification phase, in which a relatively small sparsification factor $\pi = \pi_{min}$ is used. Every time $f_{div} \cdot n$ consecutive moves do not lead to an improvement (for a given parameter f_{div}), a diversification phase is entered. Because we also solve large-scale instances and $f_{div} \cdot n$ might exceed the overall limit η_{max} of iterations without improvement, we further impose that the diversification phase is entered no later than after a given number of η_{latest} iterations without improvement. At the beginning of the diversification phase, the solution is reset to the current best solution, and the generator arc list $\mathscr{A}_g$ is *rebuilt*, i.e., all arcs selected by the sparsification method and all previously added additional arcs (see next section) are removed, and the generator arc list is restored with a higher sparsification factor π_{max}. The diversification phase spans η_{div} consecutive iterations after which the generator arc list is rebuilt with π_{min}. In addition, the generator arc list is cleared of the arcs that were added as additional arcs (see next section) every $2 \cdot n$ iterations.

Compared to the first variant, the second variant adapts the size of the generator arc list more frequently and in smaller steps, and the factors π_{min} and π_{max} serve as lower and upper bounds. The method also starts with a small sparsification factor of $\pi = \pi_{min}$. Every $\eta_{rebuild}$ iterations (called a *period*), it updates the sparsification factor π and rebuilds the generator arc list $\mathscr{A}_g$. Every new best solution also triggers a rebuild. We decrease (increase) the sparsification by the factor Δ_π if $C(x_{last})$ of the best solution x_{last} of the last period is (not) smaller than the value $C(x_{curr})$ of the best solution x_{curr} of the current period. The effect is an intensification

(diversification) of the search at times when (no) improving neighbors can be found. The update is conducted as follows:

$$\Delta_\pi = \exp\left(\frac{\log\left(\frac{\pi_{max}}{\pi_{min}}\right)}{\eta_\pi}\right) \tag{5a}$$

$$\pi = \begin{cases} \max\left(\pi_{min}, \frac{\pi}{\Delta_\pi}\right), & \text{if } C(x_{curr}) < C(x^\circ_{last}) \\ \min\left(\pi_{max}, \pi \cdot \Delta_\pi\right), & \text{if } C(x_{curr}) > C(x^\circ_{last}) \\ \pi, & \text{otherwise} \end{cases} \tag{5b}$$

3.5.4 Additional arcs

Several papers have found that inserting additional arcs into the generator arc list is beneficial to the performance of the GTS (see, e.g., Toth and Vigo 2003, Schneider et al. 2017). In case they are not already included, we add to the generator arc list $\mathscr{A}_g$:

A: the inserted arcs of every accepted neighbor solution x°, and
B: the arcs of the currently overall best solution x_{best} found by GTS.

Variant `base` does *not* include additional arcs. The alternative setting that we investigate treats arcs of type A and B differently: arcs of type A are added to $\mathscr{A}_g$ in each iteration and are inserted uniformly into random positions. Note that often the sparsification method sorts a large share of them to the end of the generator arc list, which could negatively affect the performance of the GTS in case of a first-improvement search. On the contrary, the sparsification method tends to assess customer and depot arcs of type B as promising, so that they can be inserted according to the sorting.

3.5.5 Selection of depot arcs

Toth and Vigo (2003) and Schneider et al. (2017) stress the importance of including a sufficient number of depot arcs into the generator arc list even if some depot arcs are relatively long. We consider four different variants, which initially sort depot arcs and customer arcs in individual lists (called $\mathscr{A}^d_{cand}$ and $\mathscr{A}^c_{cand}$) using the same sparsification method but differ in the decision on how many and which of the arcs of $\mathscr{A}^d_{cand}$ and $\mathscr{A}^c_{cand}$ are merged into the generator arc list $\mathscr{A}_g$. After the merge, all four variants sort $\mathscr{A}_g$ according to the sparsification criterion.

In variant `base`, customer and depot arcs are treated identically, i.e., the same sparsification factor is used to select the arcs from $\mathscr{A}^d_{cand}$ and $\mathscr{A}^c_{cand}$. The second variant puts more emphasis on the depot arcs by selecting the arcs from $\mathscr{A}^d_{cand}$ with a larger sparsification factor than those from $\mathscr{A}^c_{cand}$. The third variant is tailored to the MDVRP and guarantees a minimum number of depot arcs for each customer. More precisely, the same sparsification factor is used to select the arcs from $\mathscr{A}^d_{cand}$ and $\mathscr{A}^c_{cand}$, but we continue to select arcs from $\mathscr{A}^d_{cand}$ until the guaranteed minimum number is reached. The last variant inserts all depot arcs into $\mathscr{A}_g$.

The depot arcs are likely to appear more at the end of the sorted generator arc list because they tend to be more expensive with regard to the sparsification criterion. This might have negative effects on the quality of the GTS when using first-improvement search. Therefore, we evaluate two additional variants that use alternative sorting strategies: The first one inserts arcs from $\mathscr{A}^d_{cand}$ and $\mathscr{A}^c_{cand}$ in alternating fashion. As soon

as we have reached the threshold determined by the sparsification factor in one of the two lists, we append the remaining arcs (up to the threshold) of the second list. In the second setting, we compute the number of entries of the two lists after the sparsification factor is used to determine the threshold, i.e., we compute $|\mathscr{A}^d_{cand}(\pi_d)|$ and $|\mathscr{A}^c_{cand}(\pi_c)|$, and we uniformly distribute the entries into $\mathscr{A}_g$. The uniform distribution into the new list, which has $|\mathscr{A}^c_{cand}(\pi_c)|+|\mathscr{A}^d_{cand}(\pi_d)|$ entries in total, is established as follows: First, we assign the elements of the smaller list, i.e., of $\mathscr{A}^c_{cand}(\pi_c)$ or $\mathscr{A}^d_{cand}(\pi_d)$, to positions in the new list. More precisely, the ith element is assigned to position $\lfloor i \cdot (|\mathscr{A}^c_{cand}(\pi_c)|+|\mathscr{A}^d_{cand}(\pi_d)|)/\min(|\mathscr{A}^c_{cand}(\pi_c)|,|\mathscr{A}^d_{cand}(\pi_d)|)\rceil$ (rounding with $\lfloor\cdot\rceil$ to the next integer). The elements of the longer list are then inserted into the free positions keeping their order.

3.6 Gain-based pruning

In addition to the speedup of granular neighborhoods, we use so-called gain-based pruning to exit unprofitable move evaluations early. The idea of gain-based pruning is to stop a move evaluation as soon as we are sure that the investigated move is worse than the best found move of the current iteration.

If the pure routing cost $c(x')$ of a solution x' at hand already exceeds $C(x^\circ)$ of the best solution x° in this exploration, the computation of $C(x')$ can be stopped early. Pretests have shown that discarding those moves leads to a substantial speedup, especially if the order in which the moves are evaluated allows to find good moves early. Note that this speedup does not affect the search trajectory and therefore not the solution quality.

3.7 Continuous Diversification

Pretests have shown that the *continuous diversification* strategy proposed by Cordeau et al. (1997) is able to improve our GTS. For each arc, this diversification mechanism counts how often (*divscore*) the respective arc has been inserted into a solution. If no improving solution can be found, each move is penalized as follows:

$$C_{div}(x) = \left(1+\sigma\cdot\sum_{a\in\mathscr{A}_{ins}} divscore(a)\right)\cdot C(x), \tag{6}$$

where $\mathscr{A}_{ins}=\mathscr{A}_{ins}(x)$ denotes the set of arcs that are newly inserted into x, and the parameter σ controls the strength of the diversification. In this way, we ensure to at least explore new regions of the solution space at times at which we cannot improve. The *divscore* of all arcs is reset to zero when a new best solution is found.

4 Computational experiments

We describe the general study design (Section 4.1), the benchmark instances (Section 4.2), and the training and test sets (Section 4.3). In Section 4.4, we report the computational results and deduce the rule-based design of our GTS. The two final GTS configurations are defined in Section 4.5, and the final performance evaluations on the test set and the original benchmark sets are conducted in Section 4.6.

4.1 Study design

The aim of the following computational experiments is to derive the design of a powerful GTS through the selection of the right algorithmic components. A full-factorial testing, i.e., evaluating each possible combination of design decisions is not feasible due to the immense number of resulting configurations. Because extensive parameter tuning (as done in many evaluations of metaheuristics) bears the risk of observing overfitted results (Gendreau 2003), our study uses structurally different instances that are divided into a training set and a test set. On the training set, we compare variant `base` (mainly the GTS of Toth and Vigo (2003) with additional algorithmic decisions found in pretests) with variants that modify exactly one component of variant `base` as described in Sections 3.3–3.7. In this way, we can directly observe the influence of each design decision and assess its importance. An overview of the GTS parameters that are common to all variants is presented in Table 1 and the features of different variants in Table 16 in Appendix A.

We have chosen an experimental design that covers the three most important MDVRP benchmark sets from the literature (see Section 4.2), and we divide these 57 instances into 17 instances for training and 40 instances for the final tests (Section 4.3). The design decisions are deduced solely from experiments with the training set. For each MDVRP instance and GTS variant, we perform 10 independent runs with different random seeds. To compare the modified GTS variants with variant `base`, we use objective values and the computation times as key indicators for measuring efficacy:

- the most important indicator is the average gap $\bar{\Delta}$ (in percent, taken over the 10 independent runs) to the *best-known solution* (BKS) of the respective instance taken from Gauthier and Irnich (2020); this indicator reflects the overall solution quality and robustness of the GTS variant;
- the best gap Δ (in percent, best taken from the 10 independent runs) to the BKS of the respective instance;
- the speed factor f, i.e., the ratio of the runtimes consumed by a variant compared to the runtimes of variant `base` (the speed factor f is better suited than an absolute time in experiments in which runtimes vary strongly between instances because large values are overly represented in averages leading to distorted results);
- the average computation time t (in seconds) provides an estimate of how long a single run takes.

Because these measures are typically not normally distributed, we apply the non-parametric WSRT to compare two different variants. As a non-parametric test, the WSRT does not rely on normally distributed input data; it is however recommended to use input data that is not too skewed. Note that the average gaps $\bar{\Delta}$ to the best known solutions from the literature are indeed less skewed compared to objective values (see also, e.g., Coffin and Saltzman 2000).

Table 1: Parameters for all tested GTS variants

Component	Section	Parameter(s)
tabu tenure	3	$[\tau_{min}, \tau_{max}] = [10, 30]$
stopping criteria	3	$t_{max} = 1500s$ and $\eta_{max} = 12000$
penalties	3.1	$[\rho_{min}, \rho_{max}] = [0.01, 100]$, $\rho_{init} = 1$, and $\eta_{pen} = 100$
construction	3.2	$[cnstr_l, cnstr_u] = [0.5, 0.9]$
(dynamic) sparsification	3.5.3	$[\pi_{min}, \pi_{max}] = [0.03, 0.1]$, $\eta_{rebuild} = 50$, $\eta_{\pi} = 5$, $f_{div} = 15$, $\eta_{latest} = 9000$, and $\eta_{div} = n$
continuous diversification	3.7	$\sigma = 0.0001$

The sparsification factor π is a powerful leverage to arrive at statistically significant results for GTS algorithms: the smaller we choose π, the shorter the generator arc list $\mathscr{A}_g$ and the more important the correct selection of all algorithmic components. Therefore, we start our experimental analysis with a relatively small sparsification factor of $\pi = 0.03$, i.e., only 3% of all arcs are included in $\mathscr{A}_g$ before additional arcs are added, see Section 3.5.4. Section 4.4 presents the results of the experiments that systematically investigate the performance of different alternatives for modifying one component of variant `base` and derives design recommendations for each of the algorithmic components based on this.

We define clear *rules* on how to act on the results of the experiments:

R1: The necessary condition for accepting a design decision and integrating it into the final GTS configurations is that it either significantly improves the average gap $\bar{\Delta}$ compared to variant `base` (tested with the WSRT) or that all four key indicators ($\bar{\Delta}$, Δ, t, and f) of the variant dominate the values of *all* other variants.

R2: In case there are two or more alternative variants that fulfill the above requirement, we choose the one that provides the best tradeoff with respect to the best gap Δ and the speed factor f. The (minimal) product $\Delta \cdot f$ is a simple but reasonable indicator for the (best) tradeoff. Note that for computing this product, one first has to aggregate Δ and f over the instances of the training set.

We distinguish between GTS variants (considering single modifications in isolation) and GTS configurations (combined design decisions). We start with GTS variants and design aspects that are more isolated and have only little interdependency with other algorithmic components. With the rule-based process, we then obtain two powerful GTS configurations that address the general tradeoff between computational effort and solution quality, i.e., the first configuration called `speed` focuses on speed and the second configuration `quality` focuses more on quality (Section 4.5).

4.2 Benchmark instances and computational environment

We briefly describe the three most important MDVRP benchmark sets from the literature, which we use in our experiments. The first set was proposed by Cordeau et al. (1997) and consists of 33 instances. Instances 1–7 originate from Christofides and Eilon (1969), instances 8–11 from Gillett and Johnson (1976), instances 12–23 from Chao et al. (1993), and instances 24–33 are self-generated by Cordeau et al. (1997) and feature customer clusters. The number of customers ranges from 50 to 360 and the number of depots from 2 to 9. The vehicle fleet is homogeneous. The number of vehicles per depot varies between 1 and 14. Instances 8–11, 13, 14, 16, 17, 19, 20, 22, and 23 have limited route durations.

The second benchmark set was originally introduced for the MDVRP with time windows in Vidal et al. (2013) and converted to an MDVRP instance set by disregarding the time-window and fleet-size constraints in Vidal et al. (2014). The same generation procedure as in Cordeau et al. (1997, 2001) was employed. As in Gauthier and Irnich (2020), we use the 14 instances pr11a to pr24a with 360 to 960 customers and 4 to 12 depots.

The third benchmark set of De Smet et al. (2017) comprises real-world instances from Belgium. To compare with results of Gauthier and Irnich (2020), we choose the same 10 instances with 50 to 2750 locations that they use. All instances have a fleet size limit.

In the remainder of the paper, we refer to Cordeau et al. (1997), Vidal et al. (2014), and De Smet et al. (2017) instances simply as "Cordeau et al.", "Vidal et al.", and "De Smet et al." instances, respectively. In total, our benchmark set consists of 57 instances with different characteristics.

The GTS is implemented in `C++` and compiled into 64-bit single-thread code with `g++` version 9 using the '`-Ofast`' option. All tests are performed on the `c18m` partition of the RWTH Aachen computer cluster, which contains 1250 nodes of the type Intel HNS2600BPB (Xeon Platinum 8160 'SkyLake'). Every node has 192 GB of memory and contains 48 cores with processors clocked at 2.1 GHz. We run single-threaded exclusive jobs. The linear program (4) to obtain the reduced costs is solved using Gurobi 8.0.0.

4.3 Training and test set

In the training phase, we select the components of the GTS that result in the two configurations `speed` and `quality`. The training set comprises a representative subset of all three benchmarks instance sets. As the number of customers is the most important feature of an MDVRP instance, we further partition the test set into three subsets with small (up to 200 customers), medium (200 to 500 customers), and large instances (more than 500 customers). To obtain the final training set, we then randomly select 30% of the instances from each category. The remaining instances form the test set. Table 2 shows the resulting training and test set.

Table 2: Training set (highlighted in **bold**) and test set

	Cordeau et al.	Vidal et al.	De Smet et al.
small	p01 p12 pr01 p02 p13 **pr02** p03 p14 pr03 p04 **p15** pr04 p05 p16 pr07 **p06** **p17** pr08 p07		belgium-d2-n50-k10 **belgium-d3-n100-k10** **belgium-n50-k10p01** belgium-n100-k10p02
medium	**p08** p19 pr05 **p09** p20 pr06 p10 p21 pr09 p11 p22 pr10 p18 p23	pr11a pr12a **pr17a** **pr21a**	**belgium-d5-n500-k20** **belgium-n500-k20**
large		pr13a pr19a pr14a **pr20a** pr15a **pr22a** **pr16a** pr23a pr18a pr24a	belgium-d8-n1000-k20 **belgium-d10-n2750-k55** **belgium-n1000-k20** belgium-n2750-k55
# training # test	6 instances 27 instances	5 instances 9 instances	6 instances 4 instances

For the statistical evaluation, we use the one-tailed WSRT with null hypothesis H_0 that the performance of variant `base` is at least as good as the one of the respective modified variant. (The alternative hypothesis H_a is that the modified variant performs better than variant `base` in a consistent manner across the training set.) As regularly done in literature, we set the significance level to 5% (rejecting H_0 for p-values smaller than 0.05). In the tables reporting experimental results, we put an asterisk * to indicate that the WSRT of the particular category is significant with level $p < 0.05$ (and ** if significant with $p < 0.01$). We remind the reader that this means that the underlying sample is significant, while the table shows *aggregated* values per sample (the aggregates are not statistically tested). Finally, we point out that the Friedman test is not appropriate for our

experimental setup because we compare all variants against `base` and do not try to identify a best variant among a set of variants.

4.4 Computational results and rule-based design

The following experiments investigate the effect of the different algorithmic components presented in Section 3.

4.4.1 Results for neighborhoods

In two separate experiments, we compare the performance of GTS variants that differ in the included neighborhoods, first for string neighborhoods and second for split neighborhoods (see Section 3.3). The results of both experiments are summarized in Table 3.

Table 3: Performance measures of GTS variants using different neighborhoods

Variant	$\bar{\Delta}$ [%]	Δ [%]	f	t [s]
`base`	2.78	1.06	1.00	177.57
`strel`	*2.12	0.70	1.26	207.37
`stex`	**2.02	*0.81	1.69	258.63
`streli`	*2.31	0.81	1.33	210.39
`stexi`	**1.98	0.83	1.70	245.59
`all-strings`	**1.55	**0.32	2.82	334.09

(a) Inclusion of string neighborhoods

Variant	$\bar{\Delta}$ [%]	Δ [%]	f	t [s]
`base`	2.78	1.06	1.00	177.57
`split`	**1.90	*0.62	1.12	168.68

(b) Inclusion of split neighborhoods

The positive effect of including additional string neighborhoods is reflected in the numbers presented in Table 3a. The variant `strel`, which only includes string-relocate in addition to the classical operators, is selected based on Rules R1 and R2. We note that for all modified variants, the improvement in solution quality is substantial, and the WSRT confirms the superiority over variant `base` with high confidence (p-value < 0.05). When aiming for solution quality, the best variant is the inclusion of all four string operators (`all-strings`), but the additional computational effort in each iteration of the GTS is high, visible in the average runtime that increases by a factor of almost two. Variant `strel` has the smallest runtime factor in comparison.

Table 3b shows the performance that results from the inclusion of the split neighborhoods (`split`). Compared to variant `base`, the best gaps and average gaps decrease significantly (with a p-value ≈ 0.0011 for average gaps). Variant `split` is selected according to Rule R1.

Although adding the split operators increases the computational effort in each GTS iteration, the average runtime decreases slightly (note however the difference between the indicators t and f).

4.4.2 Results for exploration strategies

Table 4 shows the results comparing variant `base` (composite and best improvement) and the alternative variants, i.e., composite neighborhoods with first improvement (`comp-first`), VND-like with best improvement (`vnd-best`), and VND-like with first improvement (`vnd-first`). While there are no significant differences

in solution quality, all modified variants have significantly decreased runtime factors (p-values < 0.01). Variant `vnd-first` has the smallest runtime (saving more than 20% of the time compared to `base`) as well as the best gaps, i.e., it dominates all other variants. According to Rule R1, variant `vnd-first` is selected.

Table 4: Performance measures of variants varying the exploration strategy

Variant	$\bar{\Delta}$ [%]	Δ [%]	f	t [s]
`base`	2.78	1.06	1.00	177.57
`comp-first`	2.85	1.07	**0.72	140.26
`vnd-best`	2.78	0.89	**0.61	138.15
`vnd-first`	2.50	0.88	**0.56	133.37

The next experiment analyzes the order of the arcs in the generator arc list $\mathscr{A}_g$, i.e., the question of how generator arcs should be sorted. This is especially important for the first-improvement pivoting rule. Recall that in variant `base`, the initialization of the generator arc list starts with the selection of customer and depot arcs by means of the sparsification method and a given sparsification factor. Afterwards, the sparsified customer and depot arc lists are merged and again sorted according to the sparsification method. We slightly depart from the standard experimental setup now. As argued above, the sorting of the generator arc list is much more relevant for variants that use first improvement and therefore of minor importance for variant `base`. Consequently, and only for this experiment, we compare alternative variants against variant `vnd-first` selected before, and not against `base`.

The first alternative variant inserts arcs from sorted customer and depot arc lists in an alternating fashion into the generator arc list (`vnd-first + alternating`). The second inserts depot arcs uniformly into the sorted customer arc list (`vnd-first + uniformly`). Table 5 summarizes the comparison against the baseline `vnd-first`. Variant `vnd-first` is significantly better than variant `vnd-first + uniformly` (p=0.0247, considering average gaps $\bar{\Delta}$) but not significantly better than `vnd-first + alternating` (considering any metric). According to Rule R1, we use `vnd-first` without a modified sorting of the generator arcs.

Table 5: Performance measures of variants that modify the order of the generator arcs

Variant	$\bar{\Delta}$ [%]	Δ [%]	f	t [s]
`vnd-first`	*2.50	0.88	0.56	133.37
`vnd-first + alternating`	2.68	1.08	0.54	129.92
`vnd-first + uniformly`	3.08	1.07	0.55	132.82

*: significant WSRT comparing `vnd-first` against `vnd-first + uniformly`, not significant comparing `vnd-first` against `vnd-first + alternating`

4.4.3 Results for granular neighborhoods

The experiments focussing on granularity design decisions are divided into five paragraphs addressing the sparsification method, the sparsification level, the dynamic sparsification mechanism, the use of additional arcs, and finally the selection of depot arcs.

Sparsification method We now analyze the sparsification method with its sorting criterion. (The reader should not confuse the sorting performed by the sparsification method before selecting a share of the arcs

with the final sorting of the generator arc list, which gives the order in which selected generator arcs are traversed, see Line 7 of Algorithm 2 and Line 8 of Algorithm 3.) Table 6 shows the comparison of variant `base` and variant `red-costs` that uses the reduced costs computed from the network relaxation presented in Section 3.5.1 for sorting.

Table 6: Performance measures of variants that sort generator arcs by standard routing costs or reduced costs

Variant	$\bar{\Delta}$ [%]	Δ [%]	f	t [s]
`base`	2.78	1.06	1.00	177.57
`red-costs`	2.33	1.00	1.14	185.34

The WSRT does not show a significant improvement using variant `red-costs` in solution quality (p-value ≈ 0.42), although both solution quality measures show better values. According to Rule R1, we stick to standard routing costs for sorting the generator arc list $\mathscr{A}_g$.

We have analyzed this outcome in detail: The variant `red-costs` improves the result for one instance drastically leading to a strong decrease in the average gap. On the downside, it deteriorates solution quality for 9 of the 17 instances leading to a non-significant WSRT. It seems that the overhead of computing the network relaxation, which can be substantial especially for the large-scale instances, is not worth the effort.

Sparsification level Recall from Section 3.5.2 that variant `base` sparsifies on the whole graph level, which bears the risk that isolated customers are not connected via generator arcs. Three alternative variants guarantee a minimum number of five generator arcs for every customer (`lev-min-arcs-per-c`), every depot (`lev-min-arcs-per-d`), or every vertex (`lev-min-arcs-per-v`). Whenever the required number of outgoing generator arcs is not directly reached, we add sufficiently many arcs from the sorted list of the sparsification method. The alternative variant `lev-per-cst` directly sparsifies on a per-customer basis.

Table 7: Performance measures of variants varying the sparsification level

Variant	$\bar{\Delta}$ [%]	Δ [%]	f	t [s]
`base`	2.78	1.06	1.00	177.57
`lev-min-arcs-per-c`	*1.82	0.77	1.17	180.56
`lev-min-arcs-per-d`	2.89	1.24	1.01	177.55
`lev-min-arcs-per-v`	*1.81	0.70	1.19	180.38
`lev-per-cst`	*2.23	0.82	1.09	188.60

Table 7 shows the performance of the resulting variants in comparison to variant `base`. Variant `lev-min-arcs-per-d` is clearly inferior to `base`. The three other alternative variants (`lev-min-arcs-per-c`, `lev-min-arcs-per-v`, and `lev-per-cst`) improve solution quality significantly with p-values of 0.0346, 0.0267, and 0.0379, respectively. The product $\Delta \cdot f$ is minimal for variant `lev-min-arcs-per-v`, so we select this variant according to Rule R2.

Dynamic sparsification Table 8 shows the results for the alternative variant `new-dyn-spars` that uses the sparsification rules presented in Section 3.5.3. Unfortunately, the advanced rule does not improve solution quality and even slows down the GTS. We, therefore, use the original dynamic sparsification rule.

Table 8: Performance measures of variants that incorporate different rules for dynamic sparsification

Variant	$\bar{\Delta}$ [%]	Δ [%]	f	t [s]
`base`	2.78	1.06	1.00	177.57
`new-dyn-spars`	2.78	1.17	1.35	179.75

Additional arcs Table 9 presents aggregated results showing the impact that additional arcs in the generator arc list have on GTS performance. Variant `add-arcs` inserts the arcs of the best move and of the current best solution as described in Section 3.5.4.

Table 9: Performance measures of variants that incorporate additional arcs into the generator arc list

Variant	$\bar{\Delta}$ [%]	Δ [%]	f	t [s]
`base`	2.78	1.06	1.00	177.57
`add-arcs`	*2.17	**0.59	1.36	189.07

Variant `add-arcs` produces significantly improved solution quality ($\bar{\Delta}$) confirmed by the WSRT (p-value of 0.038). Hence, we use variant `add-arcs` in the following.

Selection of depot arcs Variant `base` sparsifies depot arcs with the same sparsification factor π as customer arcs, see Section 3.5.5. The alternative variants modify the sparsification in different ways, each increasing the number of depot arcs: Variants `more-dep-5` and `more-dep-10` rely on 5 and 10 times higher sparsification factors for depot arcs, `min-dep-for-c-3` and `min-dep-for-c-5` guarantee 3 and 5 outgoing depot arcs per customer, respectively, and `all-dep` includes all depot arcs.

Table 10: Performance measures of variants that increase the number of depot arcs in the generator arc list

Variant	$\bar{\Delta}$ [%]	Δ [%]	f	t [s]
`base`	2.78	1.06	1.00	177.57
`more-dep-5`	2.49	0.82	1.21	181.26
`more-dep-10`	**2.03	*0.76	1.44	198.23
`min-dep-for-c-3`	2.38	0.93	1.12	172.61
`min-dep-for-c-5`	*2.19	**0.67	1.21	176.27
`all-dep`	*1.84	*0.59	2.20	243.16

Table 10 summarizes the respective experiment. There is a significant improvement of solution quality ($\bar{\Delta}$) for the variants `more-dep-10` (p-value ≈ 0.0065), `min-dep-for-c-5` (p-value ≈ 0.0276), and `all-dep` (p-value ≈ 0.0123). According to Rules R1, these variants qualify for further consideration. If we strictly followed the study design, Rule R2 would select variant `min-dep-for-c-5` based on the minimum product $\Delta \cdot f$. Because this is the last granularization experiment, we can exploit that we see two variants that significantly improve quality but that differ strongly. We observe a large tradeoff in quality and computational effort comparing `min-dep-for-c-5` and `all-dep`. As a consequence, we decide to define two final GTS configurations based on the two variants `min-dep-for-c-5` and `all-dep`. The variant `min-dep-for-c-5` is used in configuration denoted as `speed`, and `all-dep` is used in configuration `quality`, which is intended to be used when solution quality is the priority. The configuration `quality` also uses all string neighbor-

hoods to put the main emphasis on solution quality. These two configurations are studied in more detail in the next section.

4.5 Final GTS configurations

Table 11 summarizes the features of configurations `speed` and `quality`.

Table 11: Overview of features of final GTS configurations; differences are highlighted in **bold**

Component	`speed`	`quality`
Neighborhoods	classical, **strel**, split operators	classical, **all string**, split operators
Exploration strategy	VND-like, first improvement	VND-like, first improvement
Order of the generator arcs	according to sparsification method	according to sparsification method
Sparsification method	original costs	original costs
Selection of depot arcs	**5 depot arcs per customer**	**all depot arcs**
Sparsification level	5 arcs per vertex	5 arcs per vertex
Additional arcs	of move and best solution	of move and best solution

Setting sparsification intensities is only meaningful after the entries and the sorting of the generator arc list are known. In Table 12, we investigate the values $(\pi_{min}, \pi_{max}) = \{(3\%, 10\%), (6\%, 20\%), (9\%, 30\%)\}$ for both algorithmic configurations.

Table 12: Performance measures of the final GTS configurations `speed` and `quality` run with different sparsification values; best values are highlighted in **bold**

	`speed`			`quality`		
(π_{min}, π_{max})	$\bar{\Delta}$ [%]	Δ [%]	t [s]	$\bar{\Delta}$ [%]	Δ [%]	t [s]
(3%, 10%)	**0.72**	**0.06**	**145.61**	0.60	0.04	**352.29**
(6%, 20%)	0.78	0.09	186.42	**0.59**	**−0.09**	398.06
(9%, 30%)	0.82	0.09	218.48	0.60	0.04	443.55

While runtimes increase with higher sparsification factors, there is no clear trend for the solution quality. We select $(\pi_{min}, \pi_{max}) = (3\%, 10\%)$ for configuration `speed`, and $(\pi_{min}, \pi_{max}) = (6\%, 20\%)$ for variant `quality`.

4.6 Performance on test set and original benchmark sets

Two additional experiments evaluate the GTS performance. Section 4.6.1 presents the results on the test set and Section 4.6.2 the comparison with other works from the literature.

4.6.1 Comparison on training and test set

To confirm that the parameter tuning does not produce an overfitted algorithm, we first compare the two GTS configurations `speed` and `quality` on the test and training set. The ratio of Cordeau et al./Vidal et al./De Smet et al. instances in the training set is 6:5:6, while it is 27:9:4 in the test set. Since the three benchmarks differ substantially, the analysis takes the ratios into account by weighing performance indicators: For example, the weight of the Cordeau et al. set is computed as the number of Cordeau et al. instances in the training set divided by the respective number in the test set.

Table 13 shows the weighted performance metrics of the configurations `speed` and `quality` on the test instances grouped into the original benchmark sets and the weighted average performance metrics. The fact that results on the test set are even better than on the training set (for three of four quality indicators) is a very strong indicator of non-overfitted algorithms.

Table 13: Performance measures of the final GTS configurations `speed` and `quality` on training and test set

				`speed`		`quality`	
	Set	benchmark set	weight	$\bar{\Delta}$ [%]	Δ [%]	$\bar{\Delta}$ [%]	Δ [%]
Average	training	mixed		0.72	0.06	0.59	−0.09
	test	Cordeau et al.	0.22	0.94	0.40	0.72	0.16
		Vidal et al.	0.56	2.51	1.74	2.12	1.42
		De Smet et al.	1.50	−0.70	−1.19	−0.23	−0.87
Weighted average				0.55	−0.74	0.99	−0.47

Detailed instance-by-instance results can be found in Table 17 in Appendix B.

4.6.2 Comparison against metaheuristics from the literature

We compare the results of configurations `speed` and `quality` to the ILS of Gauthier and Irnich (2020), the HGSADC of Vidal et al. (2012), the GTS of Escobar et al. (2014b), and the HGSADC+ of Vidal et al. (2014). Because all these works have only published best solution qualities, we also compare with respect to the best gaps (Δ), and only report average gaps ($\bar{\Delta}$) for the two GTS variants. Although Accorsi and Vigo (2020) report convincing results on some of the Cordeau et al. instances (see Table 18 in Appendix B), we do not include their algorithm in this comparison because they only considered the subset of the Cordeau et al. instances without duration constraints.

Table 14 presents the results of the comparison on an aggregate level, while detailed instance-by-instance results can be found in Tables 18–20 in Appendix B. To allow for a fair comparison of computation times, we translate the published runtimes of all methods into a common time measure, denoted as *adjusted time* t_{adj}, based on the Passmark scores (see www.passmark.com) of the processors used in the respective article. Another complication in the comparison results from the mixed parameterization of the ILS of Gauthier and Irnich (2020) for the three benchmark sets. To obtain similar runtimes as the competing algorithms, the authors granted 250 000 iterations to the ILS for the Cordeau et al. instances, 25 000 iterations for the Vidal et al. instances, and only 500 iterations for the De Smet et al. instances. Therefore, the comparison against the ILS algorithms of Gauthier and Irnich (2020) is somewhat distorted because the authors report decreasing runtimes for benchmarks with an increasing number of customers.

On the Cordeau et al. benchmark, we can compare our GTS configurations to the algorithms of Gauthier and Irnich (2020), Vidal et al. (2012), and Escobar et al. (2014b). Both configurations `speed` and `quality` dominate the ILS of Gauthier and Irnich (2020) because they achieve better solution quality with lower computational effort. Compared to the HGSADC of Vidal et al. (2012), both configurations are faster (`speed` approximately ten times faster) but they cannot match the solution quality. The GTS of Escobar et al. (2014b) dominates configuration `speed` but not `quality` because the latter achieves a slightly better solution quality.

On the Vidal et al. benchmark, both GTS configurations achieve better solution quality than the ILS of Gauthier and Irnich (2020) but require larger runtimes. Both configurations are clearly dominated by the HGSADC+ of Vidal et al. (2014).

Table 14: Performance measures of `speed` and `quality` compared to the algorithms of Gauthier and Irnich (2020), Vidal et al. (2012), Escobar et al. (2014b), and Vidal et al. (2014) on the benchmark sets from the literature

	`speed`			`quality`			Gauthier and Irnich (2020)		Vidal et al. (2012)		Escobar et al. (2014b)		Vidal et al. (2014)	
Processor type		Platinum 8160					Intel i7-6700		Intel Pentium IV		Intel Core Duo		Opteron 275	
Processor speed		2.1 GHz					3.40 GHz		3.00 GHz		2.00 GHz		2.20 GHz	
Passmark score		1984					2306		563		583		445	
Time factor		10 x 1					1 x 1.162		10 x 0.284		1 x 0.294		10 x 0.224	
Benchmark	Δ [%]	$\bar{\Delta}$ [%]	t_{adj} [s]	Δ [%]	$\bar{\Delta}$ [%]	t_{adj} [s]	Δ [%]	t_{adj} [s]	Δ [%]	t_{adj} [s]	Δ [%]	t_{adj} [s]	Δ [%]	t_{adj} [s]
Cordeau et al.	0.40	0.95	69.6	0.18	0.72	400.4	0.46	449.64	0.08	721.76	0.19	40.79	–	–
Vidal et al.	1.66	2.39	1133.2	1.36	2.02	6947.9	2.91	648.26	–	–	–	–	0.02	486.86
De Smet et al.	−1.36	−0.76	3584.0	−1.25	−0.45	5704.3	0.06	22.78	–	–	–	–	–	–

On the De Smet et al. benchmark, the configurations are compared to the ILS of Gauthier and Irnich (2020). These two solution approaches constitute extreme representatives with respect to the quality-time tradeoff. As one can expect, our GTS configurations take substantially more time than the 500-iteration ILS of Gauthier and Irnich (2020), which is by more than factor 100 faster than the GTS configurations `speed` and `quality`. However, both GTS configurations clearly improve solution quality: they find new best-known solutions for all except one instance (see Table 20 in Appendix B). Interestingly, configuration `quality` achieves lower solution quality than `speed`. The reason is that `quality` performs too few iterations on the large-scale instances because the large number of utilized neighborhood operators and the higher sparsification factor lead to a slower neighborhood exploration.

5 Conclusion

In this paper, we design two GTS configurations that compare reasonably with the state-of-the-art metaheuristics for the MDVRP. The two algorithms show that a rule-based algorithm design using statistical tests is able to derive algorithms with a decent performance although only well-known and simple algorithmic components are used. Moreover, the quality-time tradeoff can be adequately controlled with the GTS configurations.

For several important choices for algorithmic components, the rule-based process provides statistically supported evidence (through WSRTs):

- using string operators (especially string-relocate) and split neighborhood operators improves solution quality;
- instead of exploring a composite neighborhood, a VND-like neighborhood exploration utilizing a first-improvement (pivoting) strategy accelerates the GTS while not impeding solution quality;
- sorting the generator arc list based on reduced costs does not improve solution quality;
- including arcs connecting isolated vertices in the generator arc list is beneficial; guaranteeing a minimum number of arcs per vertex is one possibility that increases runtimes only slightly;
- the inclusion of more depot arcs in the generator arc lists improves the performance of the GTS; a minimum number of depot arcs per customer (instead of inserting all depot arcs as suggested by Toth and Vigo 2003) keeps the increase in runtime reasonable.

We conclude with Table 15 that compares the designs of GTS for different types of VRPs derived in systematic studies on the algorithmic components of GTS. For the general building blocks, Schröder et al. (2020) and the paper at hand arrive at rather similar designs: neighborhoods should be combined in a VND-like manner and neighborhood exploration should be performed with a first improvement strategy. Moreover, the inclusion of the string-relocate neighborhood turns out to be beneficial. With a focus on high-quality solutions, all neighborhoods should be included.

The recommendations regarding specific granularization aspects are less clear cut if one compares the works of Schneider et al. (2017) and Schröder et al. (2020) with our MDVRP results. It remains unclear whether different types of VRPs really require problem-tailored granularization methods or these design choices are of minor importance. We encourage other researchers to contribute with dedicated studies on local search-based metaheuristics to clarify this and other questions.

Finally, a straightforward and clear rule-based process for designing heuristics might also be used for other VRP variants and other metaheuristic paradigms in the future.

Table 15: Design recommendations for GTS derived in (Schneider et al. 2017), (Schröder et al. 2020), and this paper

	Schneider et al. (2017) VRPTW GTS	Schröder et al. (2020) CVRP pure granular LS	this paper MDVRP GTS
Exploration strategy		VND	VND-like
Pivoting rule		first improvement and variations	first improvement
Neighborhood selection		classical + string-relocate * / all	classical + string-relocate + split / all
Granularization intensity		strong to medium	strong to medium
Specific granularization decisions:			
○ depot arc distribution in generator arc list	(irrelevant, because of best improvement)		according to sparsification method
○ sparsification method	reduced cost		cost
○ minimum number of arcs	per vertex		per vertex
○ number of depot arcs	tradeoff: more depot arcs quality: all depot arcs		`speed`: min. number per vertex `quality`: all depot arcs
○ additional arcs	best (move and) solution	•	best move and best solution
○ dynamic sparsification	recommended	•	as in Toth and Vigo (2003)
○ advanced sparsification method	not recommended		

Note: Blank entries mean 'not investigated'; • = used but not investigated; * = split operators were not considered

Acknowledgment

This research was funded by the Deutsche Forschungsgemeinschaft (DFG) under grants no. IR 122/7-2 and SCHN 1497/1-2. We are grateful to RWTH Aachen University for the granted computing resources under project rwth0335.

References

L. Accorsi and D. Vigo. A hybrid metaheuristic for single truck and trailer routing problems. *Transportation Science*, 54(5):1351–1371, 2020.

R. Baldacci and A. Mingozzi. A unified exact method for solving different classes of vehicle routing problems. *Mathematical Programming*, 120(2):347–380, 2009.

M. O. Ball, B. Golden, A. Assad, and L. Bodin. Planning for truck fleet size in the presence of a common-carrier option. *Decision Sciences*, 14(1):103–120, 1983.

W. J. Bell, L. M. Dalberto, M. L. Fisher, A. J. Greenfield, R. Jaikumar, P. Kedia, R. G. Mack, and P. J. Prutzman. Improving the distribution of industrial gases with an on-line computerized routing and scheduling optimizer. *Interfaces*, 13(6):4–23, 1983.

G. G. Brown, C. J. Ellis, G. W. Graves, and D. Ronen. Real-time, wide area dispatch of mobil tank trucks. *Interfaces*, 17(1):107–120, 1987.

P. Cassidy and H. Bennett. Tramp—a multi-depot vehicle scheduling system. *Journal of the Operational Research Society*, 23(2):151–163, 1972.

I.-M. Chao, B. L. Golden, and E. Wasil. A new heuristic for the multi-depot vehicle routing problem that improves upon best-known solutions. *American Journal of Mathematical and Management Sciences*, 13(3-4):371–406, 1993.

N. Christofides and S. Eilon. An algorithm for the vehicle-dispatching problem. *Journal of the Operational Research Society*, 20(3):309–318, 1969.

M. Coffin and M. J. Saltzman. Statistical analysis of computational tests of algorithms and heuristics. *INFORMS Journal on Computing*, 12(1):24–44, 2000.

C. Contardo and R. Martinelli. A new exact algorithm for the multi-depot vehicle routing problem under capacity and route length constraints. *Discrete Optimization*, 12:129–146, 2014.

J.-F. Cordeau and M. Maischberger. A parallel iterated tabu search heuristic for vehicle routing problems. *Computers and Operations Research*, 39(9):2033–2050, 2012.

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

J.-F. Cordeau, G. Laporte, and A. Mercier. A unified tabu search heuristic for vehicle routing problems with time windows. *Journal of the Operational research society*, 52(8):928–936, 2001.

G. De Smet et al. *OptaPlanner VRP examples: Belgium 2017 dataset*. Red Hat and the community, 2017. URL https://www.optaplanner.org. OptaPlanner is an open source constraint satisfaction solver in Java.

J. W. Escobar, R. Linfati, M. G. Baldoquin, and P. Toth. A granular variable tabu neighborhood search for the capacitated location-routing problem. *Transportation Research Part B: Methodological*, 67:344–356, 2014a.

J. W. Escobar, R. Linfati, P. Toth, and M. G. Baldoquin. A hybrid granular tabu search algorithm for the multi-depot vehicle routing problem. *Journal of Heuristics*, 20(5):483–509, 2014b.

M. L. Fisher, A. J. Greenfield, R. Jaikumar, and J. T. Lester III. A computerized vehicle routing application. *Interfaces*, 12(4):42–52, 1982.

B. Funke, T. Grünert, and S. Irnich. Local search for vehicle routing and scheduling problems: Review and conceptual integration. *Journal of Heuristics*, 11(4):267–306, 2005.

J. B. Gauthier and S. Irnich. Inter-depot moves and dynamic-radius search for multi-depot vehicle routing problems. Technical Report LM-2020-03, Chair of Logistics Management, Gutenberg School of Management and Economics, Johannes Gutenberg University Mainz, Mainz, Germany, 2020.

M. Gendreau. An introduction to tabu search. In F. Glover and G. A. Kochenberger, editors, *Handbook of Metaheuristics*, pages 37–54. Springer US, Boston, MA, 2003. ISBN 978-0-306-48056-0.

M. Gendreau, A. Hertz, and G. Laporte. A tabu search heuristic for the vehicle routing problem. *Management Science*, 40(10):1276–1290, 1994.

B. E. Gillett and J. G. Johnson. Multi-terminal vehicle-dispatch algorithm. *Omega*, 4(6):711–718, 1976.

F. Glover. Tabu search—part I. *ORSA Journal on Computing*, 1(3):190–206, 1989.

D. Kirchler and R. Wolfler Calvo. A granular tabu search algorithm for the dial-a-ride problem. *Transportation Research Part B: Methodological*, 56:120–135, 2013.

N. Labadie, R. Mansini, J. Melechovský, and R. Wolfler Calvo. The team orienteering problem with time windows: An LP-based granular variable neighborhood search. *European Journal of Operational Research*, 220(1):15–27, 2012.

H. C. Lau, T. Chan, W. Tsui, and W. Pang. Application of genetic algorithms to solve the multidepot vehicle routing problem. *IEEE Transactions on Automation Science and Engineering*, 7(2):383–392, 2009.

B. Ombuki-Berman and F. T. Hanshar. Using genetic algorithms for multi-depot vehicle routing. In F. B. Pereira and J. Tavares, editors, *Bio-inspired Algorithms for the Vehicle Routing Problem*, pages 77–99. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009. ISBN 978-3-540-85152-3.

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

J. Pooley. Integrated production and distribution facility planning at ault foods. *Interfaces*, 24(4):113–121, 1994.

C. Prins, C. Prodhon, A. Ruiz, P. Soriano, and R. Wolfler Calvo. Solving the capacitated location-routing problem by a cooperative Lagrangean relaxation-granular tabu search heuristic. *Transportation Science*, 41(4):470–483, 2007.

J. Renaud, G. Laporte, and F. F. Boctor. A tabu search heuristic for the multi-depot vehicle routing problem. *Computers and Operations Research*, 23(3):229–235, 1996.

R. Sadykov, E. Uchoa, and A. Pessoa. A bucket graph–based labeling algorithm with application to vehicle routing. *Transportation Science*, 55(1):4–28, 2021.

M. Schneider and M. Löffler. Large composite neighborhoods for the capacitated location-routing problem. *Transportation Science*, 53(1):301–318, 2019.

M. Schneider, F. Schwahn, and D. Vigo. Designing granular solution methods for routing problems with time windows. *European Journal of Operational Research*, 263(2):493–509, 2017.

C. Schröder, M. Schneider, J. B. Gauthier, and T. Gschwind. In-depth analysis of granular local search for capacitated vehicle routing. Working Paper dpo-2020-03, Deutsche Post Chair – Optimization of Distribution Networks, RWTH Aachen University, Aachen, Germany, 2020.

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

S. R. Thangiah and S. Salhi. Genetic clustering: an adaptive heuristic for the multidepot vehicle routing problem. *Applied Artificial Intelligence*, 15(4):361–383, 2001.

P. Toth and D. Vigo, editors. *The Vehicle Routing Problem*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2002. ISBN 0-89871-498-2.

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

T. Vidal, T. G. Crainic, M. Gendreau, N. Lahrichi, and W. Rei. A hybrid genetic algorithm for multidepot and periodic vehicle routing problems. *Operations Research*, 60(3):611–624, 2012.

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

T. Vidal, T. G. Crainic, M. Gendreau, and C. Prins. Implicit depot assignments and rotations in vehicle routing heuristics. *European Journal of Operational Research*, 237(1):15–28, 2014.

D. Vigo and P. Toth, editors. *Vehicle Routing*. MOS-SIAM Series on Optimization. Society for Industrial and Applied Mathematics, Philadelphia, PA, 2014. ISBN 978-1-61197-358-7.

F. Wilcoxon. Individual comparisons by ranking methods. *Biometrics Bulletin*, 1(6):80–83, 1945.

A Features of different GTS settings

Table 16: Overview of features of all GTS variants

Component	`base`	`all-strings`	`split`	`red-costs`	`comp-first`	`vnd-best`	`vnd-first`	`lev-min-arcs-per-c`	`lev-min-arcs-per-d`	`lev-min-arcs-per-v`	`lev-per-cst`
Neighborhood operators:											
- four classical operators	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
- all string operators		✓									
- only one string operator											
- split operators			✓								
Sparsification method:											
- original costs	✓	✓	✓	✗	✓	✓	✓	✓	✓	✓	✓
- reduced costs				✓							
Exploration strategy:											
- composite	✓	✓	✓	✓	✓	✗	✗	✓	✓	✓	✓
- VND-like						✓	✓				
- best improvement	✓	✓	✓	✓	✗	✓	✗	✓	✓	✓	✓
- first improvement					✓		✓				
Addressing isolated vertices:											
- whole graph spars. (no a.)	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✗
- min. c. arcs per c.								✓		✓	
- min. c. arcs per d.									✓	✓	
- per-customer											✓
Depot arc sparsification:											
- normal sparse factor	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
- sparse factor times 5											
- sparse factor times 10											
- all											
- min. 3 d. arcs per c.											
- min. 5 d. arcs per c.											
Additional arcs:											
- best move's inserted arcs											
- best current solution arcs											
Order of gen. arc list:											
- acc. to sparsification m.	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
- alternating											
- uniformly											

Checks (✓) indicate chosen features, gray checks (✓) chosen features that are also present in variant `base`, and crosses (✗) features that are present in variant `base` but not in the particular configuration.

Table 16: Overview of features of all GTS variants (continued)

Component	`all-dep`	`more-dep-5`	`more-dep-10`	`min-dep-for-c-3`	`min-dep-for-c-5`	`add-arcs`	`alternating`	`uniformly`
Neighborhood operators:								
- four classical operators	✓	✓	✓	✓	✓	✓	✓	✓
- all string operators								
- only one string operator								
- split operators								
Sparsification method:								
- original costs	✓	✓	✓	✓	✓	✓	✓	✓
- reduced costs								
Exploration strategy:								
- composite	✓	✓	✓	✓	✓	✓	✓	✓
- VND-like								
- best improvement	✓	✓	✓	✓	✓	✓	✓	✓
- first improvement								
Addressing isolated vertices:								
- whole graph spars. (no a.)	✓	✓	✓	✓	✓	✓	✓	✓
- min. c. arcs per c.								
- min. c. arcs per d.								
- per-customer								
Depot arc sparsification:								
- normal sparse factor	✗	✗	✗	✓	✓	✓	✓	✓
- sparse factor times 5		✓						
- sparse factor times 10			✓					
- all	✓							
- min. 3 d. arcs per c.				✓				
- min. 5 d. arcs per c.					✓			
Additional arcs:								
- best move's inserted arcs						✓		
- best current solution arcs						✓		
Order of gen. arc list:								
- acc. to sparsification m.	✓	✓	✓	✓	✓	✓	✗	✗
- alternating							✓	
- uniformly								✓

Checks (✓) indicate chosen features, gray checks (✓) chosen features that are also present in variant `base`, and crosses (✗) features that are present in variant `base` but not in the particular configuration.

B Results of GTS configurations on test and original benchmark instances

Table 17: Performance measures of `speed` and `quality` run on the test instances

		`speed`					`quality`				
Instance	BKS	z	$\bar{z}$	t [s]	Δ [%]	$\bar{\Delta}$ [%]	z	$\bar{z}$	t [s]	Δ [%]	$\bar{\Delta}$ [%]
p01	576.87	576.87	576.87	1.12	0.00	0.00	576.87	576.87	3.45	0.00	0.00
p02	473.53	473.53	473.81	1.23	0.00	0.06	473.53	473.84	3.19	0.00	0.06
p03	641.19	641.19	641.52	1.86	0.00	0.05	641.19	642.17	5.97	0.00	0.15
p04	1001.04	1001.04	1006.40	2.68	0.00	0.54	1001.04	1003.58	14.60	0.00	0.25
p05	750.03	750.03	751.14	1.99	0.00	0.15	750.11	751.83	6.46	0.01	0.24
p07	881.97	891.62	895.58	3.08	1.09	1.54	884.66	894.59	13.76	0.31	1.43
p10	3629.60	3634.29	3687.46	10.64	0.13	1.59	3632.70	3668.41	70.14	0.09	1.07
p11	3545.18	3585.69	3613.00	10.57	1.14	1.91	3546.62	3601.11	77.32	0.04	1.58
p12	1318.95	1318.95	1319.15	1.50	0.00	0.02	1318.95	1318.95	4.64	0.00	0.00
p13	1318.95	1318.95	1318.95	1.58	0.00	0.00	1318.95	1318.95	4.86	0.00	0.00
p14	1360.12	1360.12	1361.23	1.30	0.00	0.08	1360.12	1360.12	4.69	0.00	0.00
p16	2572.23	2572.23	2578.38	3.84	0.00	0.24	2572.23	2585.04	17.50	0.00	0.50
p18	3702.85	3702.85	3722.40	10.72	0.00	0.53	3708.71	3734.35	57.79	0.16	0.85
p19	3827.06	3827.06	3848.82	6.01	0.00	0.57	3827.06	3849.34	34.66	0.00	0.58
p20	4058.07	4097.14	4112.87	8.55	0.96	1.35	4058.07	4082.74	49.34	0.00	0.61
p21	5474.84	5497.75	5547.25	22.25	0.42	1.32	5490.11	5524.87	160.49	0.28	0.91
p22	5702.16	5726.75	5743.27	13.55	0.43	0.72	5702.16	5728.88	84.69	0.00	0.47
p23	6078.75	6123.48	6187.70	19.82	0.74	1.79	6078.75	6129.66	139.83	0.00	0.84
pr01	861.32	861.32	861.32	0.65	0.00	0.00	861.32	861.32	1.55	0.00	0.00
pr03	1803.80	1805.20	1816.44	4.16	0.08	0.70	1804.59	1815.16	14.91	0.04	0.63
pr04	2058.31	2066.41	2104.34	7.18	0.39	2.24	2075.18	2109.59	35.30	0.82	2.49
pr05	2331.20	2346.91	2371.92	10.42	0.67	1.75	2333.52	2360.36	55.71	0.10	1.25
pr06	2676.30	2711.88	2739.21	18.32	1.33	2.35	2698.25	2724.51	78.51	0.82	1.80
pr07	1089.56	1089.56	1089.56	1.41	0.00	0.00	1089.56	1089.56	3.71	0.00	0.00
pr08	1664.85	1672.18	1680.81	3.44	0.44	0.96	1666.94	1671.27	12.63	0.13	0.39
pr09	2133.20	2159.33	2168.30	6.62	1.22	1.65	2142.45	2165.28	43.34	0.43	1.50
pr10	2868.20	2919.99	2959.46	18.77	1.81	3.18	2903.45	2923.37	116.35	1.23	1.92
pr11a	4994.67	5069.06	5115.26	21.15	1.49	2.41	5046.14	5100.72	124.80	1.03	2.12
pr12a	6367.67	6429.56	6465.87	44.69	0.97	1.54	6424.13	6459.62	290.54	0.89	1.44
pr13a	7645.29	7797.54	7853.22	83.30	1.99	2.72	7738.13	7826.37	447.72	1.21	2.37
pr14a	9101.67	9282.34	9376.73	102.73	1.99	3.02	9282.75	9311.19	800.48	1.99	2.30
pr15a	10598.7	10794.70	10871.00	186.07	1.85	2.57	10791.40	10816.90	1128.56	1.82	2.06
pr18a	6504.36	6556.44	6620.07	65.86	0.80	1.78	6556.99	6619.12	408.84	0.81	1.76
pr19a	8639.44	8823.52	8879.79	128.02	2.13	2.78	8770.16	8851.08	759.17	1.51	2.45
pr23a	8014.10	8204.13	8261.16	143.78	2.37	3.08	8160.70	8209.84	956.46	1.83	2.44
pr24a	9909.49	10112.10	10171.50	238.49	2.04	2.64	10080.90	10118.50	1463.10	1.73	2.11
belgium-d2-n50-k10	15.46	15.46	15.46	1.07	-0.01	0.00	15.46	15.46	2.87	-0.01	0.01
belgium-d8-n1000-k20	50.56	48.80	49.09	214.12	-3.47	-2.90	48.73	49.08	979.87	-3.61	-2.92
belgium-n100-k10	22.98	22.70	22.83	2.13	-1.20	-0.65	22.70	22.75	7.89	-1.21	-1.00
belgium-n2750-k55	128.67	128.55	129.63	1500.04	-0.09	0.75	130.41	132.53	1500.23	1.35	3.00
$\emptyset$				73.12	0.54	1.13			249.65	0.34	0.94

Table 18: Performance measures of `speed` and `quality` compared to algorithms of Gauthier and Irnich (2020), Vidal et al. (2012), Escobar et al. (2014b), and Accorsi and Vigo (2020) run on the Cordeau et al. instances

		`speed`					`quality`					Gauthier and Irnich (2020)			Vidal et al. (2012)			Escobar et al. (2014b)			Accorsi and Vigo (2020)	
Instance	BKS	z	$\bar{z}$	t [s]	Δ [%]	$\bar{\Delta}$ [%]	z	$\bar{z}$	t [s]	Δ [%]	$\bar{\Delta}$ [%]	z	t [s]	Δ [%]	z	t [s]	Δ [%]	z	t [s]	Δ [%]	t_{10} [s]	Δ [%]
p01	576.87	576.87	576.87	1.12	0.00	0.00	576.87	576.87	3.45	0.00	0.00	576.87	71.0	0.00	576.87	13.8	0.00	576.87	7	0.00	10	0.00
p02	473.53	473.53	473.81	1.23	0.00	0.06	473.53	473.84	3.19	0.00	0.06	473.53	55.7	0.00	473.53	12.6	0.00	473.53	6	0.00	10	0.00
p03	641.19	641.19	641.52	1.86	0.00	0.05	641.19	642.17	5.97	0.00	0.15	641.19	94.9	0.00	641.19	25.8	0.00	641.19	29	0.00	20	0.00
p04	1001.04	1001.04	1006.40	2.68	0.00	0.54	1001.04	1003.58	14.60	0.00	0.25	1003.59	153.1	0.25	1001.23	116.4	0.02	1001.04	90	0.00	178	0.00
p05	750.03	750.03	751.14	1.99	0.00	0.15	750.11	751.83	6.46	0.01	0.24	751.15	114.6	0.15	750.03	63.6	0.00	750.03	26	0.00	56	0.00
p06	876.50	876.50	879.80	3.18	0.00	0.38	876.51	881.30	12.01	0.00	0.55	880.04	136.8	0.40	876.50	68.4	0.00	876.50	103	0.00	75	0.00
p07	881.97	891.62	895.58	3.08	1.09	1.54	884.66	894.59	13.76	0.31	1.43	891.33	189.2	1.06	884.43	93.0	0.28	884.66	106	0.30	86	0.00
p08	4371.66	4410.32	4441.86	11.99	0.88	1.61	4391.39	4433.87	68.85	0.45	1.42	4423.40	499.7	1.18	4397.42	600.0	0.59	4371.66	285	0.00	–	–
p09	3858.66	3896.57	3940.78	10.45	0.98	2.13	3892.37	3915.96	79.79	0.87	1.48	3899.13	469.7	1.05	3868.59	570.0	0.26	3880.85	256	0.58	–	–
p10	3629.60	3634.29	3687.46	10.64	0.13	1.59	3632.70	3668.41	70.14	0.09	1.07	3668.89	440.2	1.08	3636.08	589.2	0.18	3629.60	267	0.00	–	–
p11	3545.18	3585.69	3613.00	10.57	1.14	1.91	3546.62	3601.11	77.32	0.04	1.58	3569.11	476.7	0.68	3548.25	428.4	0.09	3545.18	192	0.00	–	–
p12	1318.95	1318.95	1319.15	1.50	0.00	0.02	1318.95	1318.95	4.64	0.00	0.00	1318.95	64.3	0.00	1318.95	31.2	0.00	1318.95	6	0.00	10	0.00
p13	1318.95	1318.95	1318.95	1.58	0.00	0.00	1318.95	1318.95	4.86	0.00	0.00	1318.95	49.2	0.00	1318.95	34.2	0.00	1318.95	7	0.00	–	–
p14	1360.12	1360.12	1361.23	1.30	0.00	0.08	1360.12	1360.12	4.69	0.00	0.00	1360.12	63.0	0.00	1360.12	33.0	0.00	1360.12	6	0.00	–	–
p15	2505.42	2505.42	2508.05	4.45	0.00	0.10	2505.42	2505.42	17.22	0.00	0.00	2505.42	181.0	0.00	2505.42	115.2	0.00	2505.42	114	0.00	89	0.00
p16	2572.23	2572.23	2578.38	3.84	0.00	0.24	2572.23	2585.04	17.50	0.00	0.50	2572.23	139.2	0.00	2572.23	118.2	0.00	2572.23	118	0.00	–	–
p17	2709.09	2720.23	2743.94	4.12	0.41	1.29	2709.09	2720.26	22.24	0.00	0.41	2742.80	172.4	1.24	2709.09	128.4	0.00	2709.09	108	0.00	–	–
p18	3702.85	3702.85	3722.40	10.72	0.00	0.53	3708.71	3734.35	57.79	0.16	0.85	3702.85	351.6	0.00	3702.85	271.2	0.00	3702.85	278	0.00	273	0.00
p19	3827.06	3827.06	3848.82	6.01	0.00	0.57	3827.06	3849.34	34.66	0.00	0.58	3827.06	258.1	0.00	3827.06	252.0	0.00	3827.06	256	0.00	–	–
p20	4058.07	4097.14	4112.87	8.55	0.96	1.35	4058.07	4082.74	49.34	0.00	0.61	4091.78	308.4	0.83	4058.07	262.2	0.00	4058.07	267	0.00	–	–
p21	5474.84	5497.75	5547.25	22.25	0.42	1.32	5490.11	5524.87	160.49	0.28	0.91	5490.11	693.9	0.28	5476.41	600.0	0.03	5474.84	268	0.00	774	0.00
p22	5702.16	5726.75	5743.27	13.55	0.43	0.72	5702.16	5728.88	84.69	0.00	0.47	5702.16	487.2	0.00	5702.16	600.0	0.00	5702.16	262	0.00	–	–
p23	6078.75	6123.48	6187.70	19.82	0.74	1.79	6078.75	6129.66	139.83	0.00	0.84	6160.98	589.7	1.35	6078.75	600.0	0.00	6095.46	285	0.27	–	–
pr01	861.32	861.32	861.32	0.65	0.00	0.00	861.32	861.32	1.55	0.00	0.00	861.32	82.1	0.00	861.32	10.2	0.00	861.32	4	0.00	–	–
pr02	1307.34	1307.61	1315.06	2.08	0.02	0.59	1307.34	1313.70	5.88	0.00	0.49	1307.34	369.7	0.00	1307.34	45.6	0.00	1311.11	11	0.29	–	–
pr03	1803.80	1805.20	1816.44	4.16	0.08	0.70	1804.59	1815.16	14.91	0.04	0.63	1806.53	208.5	0.15	1803.80	114.6	0.00	1803.80	118	0.00	–	–
pr04	2058.31	2066.41	2104.34	7.18	0.39	2.24	2075.18	2109.59	35.30	0.82	2.49	2073.31	422.8	0.73	2059.36	313.2	0.05	2064.11	124	0.28	–	–
pr05	2331.20	2346.91	2371.92	10.42	0.67	1.75	2333.52	2360.36	55.71	0.10	1.25	2359.04	948.7	1.19	2340.29	573.6	0.39	2349.63	213	0.79	–	–
pr06	2676.30	2711.88	2739.21	18.32	1.33	2.35	2698.25	2724.51	78.51	0.82	1.80	2703.31	708.5	1.01	2681.93	600.0	0.21	2710.30	234	1.27	–	–
pr07	1089.56	1089.56	1089.56	1.41	0.00	0.00	1089.56	1089.56	3.71	0.00	0.00	1089.56	191.0	0.00	1089.56	20.4	0.00	1089.56	11	0.00	–	–
pr08	1664.85	1672.18	1680.81	3.44	0.44	0.96	1666.94	1671.27	12.63	0.13	0.39	1667.24	764.1	0.14	1665.05	123.0	0.01	1665.50	66	0.04	–	–
pr09	2133.20	2159.33	2168.30	6.62	1.22	1.65	2142.45	2165.28	43.34	0.43	1.50	2148.61	524.5	0.72	2134.17	366.0	0.05	2151.45	156	0.86	–	–
pr10	2868.20	2919.99	2959.46	18.77	1.81	3.18	2903.45	2923.37	116.35	1.23	1.92	2912.24	2486.7	1.54	2886.59	600.0	0.64	2910.78	302	1.48	–	–
∅				6.96	0.40	0.95			40.04	0.18	0.72		386.85	0.46		254.35	0.08		138.82	0.19	143.73	
processor type				Platinum 8160									Intel i7-6700			Intel Pentium IV			Intel Core Duo		Intel i7-8700K	
processor speed				2.1 GHz									3.40 GHz			3.00 GHz			2.00 GHz		3.7 GHz	
passmark score				1984									2306			563			583		2704	
adjusted times				10x6.96					10x40.04				1x449.64			10x72.176			1x40.79		1x195.90	

Table 19: Performance measures of `speed` compared to the algorithms of Gauthier and Irnich (2020) and Vidal et al. (2014) run on the Vidal et al. instances

		`speed`					`quality`					Gauthier and Irnich (2020)			Vidal et al. (2014)		
Instance	BKS	z	$\bar{z}$	t [s]	Δ [%]	$\bar{\Delta}$ [%]	z	$\bar{z}$	t [s]	Δ [%]	$\bar{\Delta}$ [%]	z	t [s]	Δ [%]	z	t [s]	Δ [%]
pr11a	4994.67	5069.06	5115.26	21.15	1.49	2.41	5046.14	5100.72	124.80	1.03	2.12	5129.05	122.4	2.69	4994.67	81.90	0.00
pr12a	6367.67	6429.56	6465.87	44.69	0.97	1.54	6424.13	6459.62	290.54	0.89	1.44	6555.29	212.8	2.95	6375.87	139.18	0.13
pr13a	7645.29	7797.54	7853.22	83.30	1.99	2.72	7738.13	7826.37	447.72	1.21	2.37	7917.86	343.3	3.57	7648.71	175.03	0.04
pr14a	9101.67	9282.34	9376.73	102.73	1.99	3.02	9282.75	9311.19	800.48	1.99	2.30	9403.68	474.3	3.32	9101.67	296.51	0.00
pr15a	10598.7	10794.70	10871.00	186.07	1.85	2.57	10791.40	10816.90	1128.56	1.82	2.06	10920.67	740.4	3.04	10598.70	296.53	0.00
pr16a	11919.71	12166.00	12281.20	246.99	2.07	3.03	12154.60	12204.50	1362.71	1.97	2.39	12312.47	1048.4	3.30	11921.00	300.32	0.01
pr17a	4761.70	4832.61	4849.60	21.48	1.49	1.85	4802.88	4830.49	104.59	0.86	1.44	4856.80	153.4	2.00	4761.70	73.10	0.00
pr18a	6504.36	6556.44	6620.07	65.86	0.80	1.78	6556.99	6619.12	408.84	0.81	1.76	6666.09	313.4	2.49	6504.00	162.11	0.00
pr19a	8639.44	8823.52	8879.79	128.02	2.13	2.78	8770.16	8851.08	759.17	1.51	2.45	8971.01	649.9	3.84	8639.44	296.34	0.00
pr20a	9825.5	9989.40	10064.00	181.62	1.67	2.43	9978.81	10024.30	1190.71	1.56	2.02	10160.99	1014	3.41	9825.50	300.26	0.00
pr21a	4582.62	4626.88	4652.98	39.17	0.97	1.54	4618.21	4650.89	208.95	0.78	1.49	4677.77	228.7	2.08	4586.41	88.90	0.08
pr22a	6141.63	6229.93	6266.54	83.16	1.44	2.03	6204.87	6259.61	480.51	1.03	1.92	6283.19	405.6	2.30	6141.63	233.83	0.00
pr23a	8014.10	8204.13	8261.16	143.78	2.37	3.08	8160.70	8209.84	956.46	1.83	2.44	8218.96	764.2	2.56	8014.10	294.29	0.00
pr24a	9909.49	10112.10	10171.50	238.49	2.04	2.64	10080.90	10118.50	1463.10	1.73	2.11	10221.22	1337.6	3.15	9910.02	300.57	0.01
$\emptyset$				113.32	1.66	2.39			694.79	1.36	2.02		557.74	2.91		217.06	0.02
processor type				Platinum 8160									Intel i7-6700			Opteron 275	
processor speed				2.1 GHz									3.40 GHz			2.20 GHz	
passmark score				198 4									2306			445	
adjusted times				10x113.32					10x694.79				1x648.26			10x48.686	

Table 20: Performance measures of `speed` and `quality` compared to the ILS of Gauthier and Irnich (2020) run on the De Smet et al. instances

		`speed`					`quality`					Gauthier and Irnich (2020)		
Instance	BKS	z	$\bar{z}$	t [s]	Δ [%]	$\bar{\Delta}$ [%]	z	$\bar{z}$	t [s]	Δ [%]	$\bar{\Delta}$ [%]	z	t [s]	Δ [%]
belgium-d2-n50-k10	15.46	15.46	15.46	1.07	−0.01	0.00	15.46	15.46	2.87	−0.01	0.01	15.46	0.1	0.00
belgium-d3-n100-k10	17.33	17.30	17.34	1.76	−0.20	0.03	17.30	17.30	5.84	−0.20	−0.20	17.33	0.3	0.00
belgium-d5-n500-k20	38.43	37.39	37.66	37.77	−2.71	−2.01	37.17	37.67	247.92	−3.28	−1.97	38.43	3.1	0.00
belgium-d8-n1000-k20	50.56	48.80	49.09	214.12	−3.47	−2.90	48.73	49.08	979.87	−3.61	−2.92	50.56	13.6	0.00
belgium-d10-n2750-k55	90.70	90.85	91.65	1500.05	0.16	1.04	92.34	93.57	1500.29	1.81	3.16	90.70	65.2	0.00
belgium-n50-k10	21.02	20.70	20.79	0.99	−1.53	−1.09	20.70	20.74	3.14	−1.53	−1.35	21.02	0.1	0.00
belgium-n100-k10	22.98	22.70	22.83	2.13	−1.20	−0.65	22.70	22.75	7.89	−1.21	−1.00	23.11	0.3	0.57
belgium-n500-k20	47.75	46.72	47.25	49.84	−2.17	−1.04	46.78	47.31	215.02	−2.03	−0.91	47.75	3.1	0.00
belgium-n1000-k20	59.20	57.77	58.17	276.18	−2.42	−1.74	56.97	57.85	1241.28	−3.77	−2.27	59.20	9.9	0.00
belgium-n2750-k55	128.67	128.55	129.63	1500.04	−0.09	0.75	130.41	132.53	1500.23	1.35	3.00	128.67	100.1	0.00
∅				358.40	-1.36	-0.72			570.43	-1.25	-0.45		19.58	0.06
processor type				Platinum 8160									Intel i7-6700	
processor speed				2.1 GHz									3.40 GHz	
passmark score				1984									2306	
adjusted times				10x358.40					10x570.43				1x22.76	