Solving the traveling delivery person problem with limited computational time

Jan Mikula · **Miroslav Kulich**

Received: date / Accepted: date

Abstract The *traveling delivery person problem* (TDP) is a customer-oriented modification of the *traveling salesperson problem*, which minimizes the sum of delivery times at customers' destinations. Besides classical applications in routing or emergency logistics, it has recently emerged in other research areas like mission planning in mobile robotics, where the goal is to search a known or unknown environment efficiently. In such a specific use case, the time to solve the problem is usually limited to units or tens of seconds, or a solution's usefulness depends on the time needed to obtain it. In this paper, we solve the TDP under these restrictions for the first time. We review the methodology for evaluating stochastic algorithms in less traditional applications and then apply it to create a new metaheuristic for the problem. The process of designing the metaheuristic combines systematic and empirical approaches and is described in a step-by-step fashion. Evaluated on several sets of benchmark instances, the final method significantly outperforms the current best approach from the literature under hard time limit settings with limits ranging from 1 to 100 seconds. As shown on a subset of the instances, it also provides competitive results in the traditional sense and with cost targets corresponding to the best-known solutions worsened by 1%. Analyzing different target solution costs shows that more accessible targets are found by the proposed method even faster when compared to the reference method. Moreover, the proposed method finds four new best-known solutions of 500-customer instances.

Keywords *metaheuristics · traveling delivery person problem · minimum latency problem · run-time distribution · variable neighborhood search*

J. Mikula
Aff. 1: FEE, CTU in Prague; Technická 2, 16000 Praha 6, Czech Republic
Aff. 2: CIIRC, CTU in Prague; Jugoslávských partyzánů 1580/3, 16000 Praha 6, Czech Republic
The author to whom all correspondence should be addressed.
E-mail: mikulj14@fel.cvut.cz, jan.mikula@cvut.cz
ORCID: 0000-0003-3404-8742

M. Kulich
CIIRC, CTU in Prague; Jugoslávských partyzánů 1580/3, 16000 Praha 6, Czech Republic
E-mail: miroslav.kulich@cvut.cz
ORCID: 0000-0002-0997-5889

Declarations

Funding. This work has been supported by the European Union's Horizon 2020 research and innovation program under grant agreement No. 688117, the project Rob4Ind4.0 CZ.02.1.01/0.0/0.0/15_003/0000470, and the European Regional Development Fund. The work of Jan Mikula was also supported by the Grant Agency of the Czech Technical University in Prague, grant No. SGS21/185/OHK3/3T/37.

Conflicts of interest/Competing interests. Not applicable.

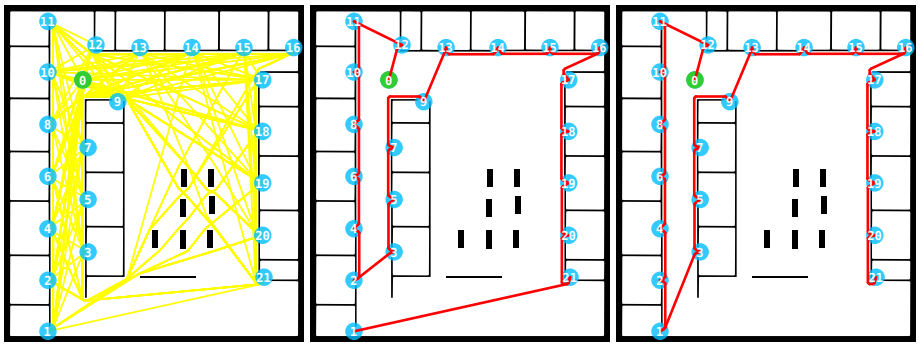
Availability of data and material. Not applicable.

Code availability. The code is available at http://imr.ciirc.cvut.cz/Downloads/Software.

1 Introduction

The traveling delivery person problem. Suppose a delivery person departing from a depot and a set of customers waiting for their deliveries. The travel times from the depot to all customers and between all pairs of the customers are known. The *traveling delivery person problem* (TDP) asks for a sequence of visits such that each customer is served (exactly once), and the sum of all customers' waiting times is minimized. The waiting time of the i-th customer in sequence $\boldsymbol{x}$ is called a latency of the delivery person at i-th customer and is denoted as $\delta_i^{\boldsymbol{x}}$. The latency is just the travel time from the depot for the first customer. However, the following customers must wait the same amount of time as the customer before them, plus the additional time required to move to their place. The delivery person seeks to satisfy the customers rather than minimize own travel expenses; therefore, the problem can be viewed as customer-oriented. A closely related and well-studied *traveling salesperson problem* (TSP) can be formulated similarly to the TDP. However, the TSP minimizes the total travel time to visit all customers, which is solely beneficial to the service provider. Thus, the TSP is so-called server-oriented (Archer and Williamson, 2003). Both problems are known to be NP-hard for general metric spaces (Sahni and Gonzalez, 1976). As their range of applications is multidisciplinary and wide, they have received much attention in the operations research literature in past decades. For an exhaustive overview of the TSP and its applications, see Cook (2012); practical applications of the TDP that are traditionally mentioned by operations research community are, e.g., customer-centered routing such as pizza-delivery or repairs of appliances, data retrieval in computer networks or emergency logistics (Fischetti et al., 1993; Ausiello et al., 2000; Campbell et al., 2008).

The TSP/TDP in mobile robotics. Recently, a different side of the scientific community started noticing these problems and seeking their efficient solutions — mobile robotics. Well-studied is, for example, the robotic variant of the TSP where the customers represent points of interest in a considered environment, and the travel time between a pair of points is proportional to a precomputed length of the shortest collision-free path connecting them (Faigl and Hollinger, 2014). An efficient solution to this problem leads to an algorithm that helps a mobile robot to effectively build a map of the environment in which it operates or to patrol an area that is a-priori known. In fact, many variants of the TSP are often considered in robotics, e.g., *TSP with neighborhoods* (Gentilini et al., 2013), *generalized*



(a) Discrete MRS instance. (b) TDP solution. (c) Another solution.

Fig. 1: How the TDP is used in mobile robotics.

TSP (Smith and Imeson, 2017), or *orienteering problem* (Gunawan et al., 2016). The TDP has also found its way to robotic problems. As shown in Kulich et al. (2014, 2017), it can be used to formulate the *mobile robot search* (MRS), another problem proven to be NP-hard in Sarmiento et al. (2004). The solution can be applied, e.g., as an efficient planner in a rescue scenario where a mobile agent searches for victims after some catastrophic event.

The TDP used to solve the MRS: toy example. Here we briefly demonstrate the exact utilization of the TDP in mobile robotics on a toy instance of a discrete MRS shown in Fig. 1. Assume an autonomous mobile robot operating in a known environment with obstacles (sized $33 \times 37\,\mathrm{m}^2$) and a task of finding a single object whose location is unknown but limited to a discrete set of possible places labeled $1, 2, \ldots, 21$ (shown in blue). The robot possesses a 2D map of the environment. It can compute (and eventually execute at a constant speed $1\,\mathrm{m/s}$) the shortest collision-free path between any two points of the environment. The shortest paths between the labeled places are shown in yellow in Fig. 1a. If the robot reaches any of the labeled places, it can report with certainty whether the object is there or not. In the MRS, the robot's goal is to visit the labeled places in such an order that it finds the searched object at the earliest on average. In a more formal language, the MRS is the problem of finding a sequence of visits $\boldsymbol{x} = (x_0 = 0, x_1, \ldots, x_n)$ (permutation of the labels; $n = 21$) starting at the robot's initial position (labeled 0, shown in green), which minimizes the expected time to find the searched object $t_{exp} = E[T|\boldsymbol{x}] = \sum_{i=1}^{n} \delta_i^{\boldsymbol{x}}\, p(x_i)$ (Sarmiento et al., 2004). Here T is a random variable taking the value of the time when the object is found, $\delta_i^{\boldsymbol{x}}$ is the latency of arriving at x_i, and $p(x_i)$ is the probability that the object is located at x_i. By assuming that all the labeled points are equally likely to exhibit the searched object, i.e., $p(k) = 1/n$, we can solve the MRS by solving the TDP (compare minimizing $E[T|\boldsymbol{x}]$ to minimizing `Cost(`$\boldsymbol{x}$`)` in Eq. (2), Sec. 3). A high-quality solution obtained by solving the TDP is shown in Fig. 1b. The solution has the total latency (sum of latencies over all visits) equal to 1524 s; therefore, the expected time $t_{exp} = 1524\,\mathrm{s}/21 \approx 72.6\,\mathrm{s}$. Note that the time to find the object in the worst possible case is when the object is located at x_n and is thus equal to the total time of travel $t_{tot} = 160\,\mathrm{s}$ along the sequence $\boldsymbol{x}$. Further note that minimizing the expected time t_{exp} and the worst time t_{tot} is not the same, which illustrates another solution in Fig. 1c. Here the displayed sequence has $t_{exp} = 1561\,\mathrm{s}/21 \approx 74.3\,\mathrm{s}$ (this solution is worse on average), but it also has $t_{tot} = 146\,\mathrm{s}$ (this solution is better in the worst case).

Planning in mobile robotics: why is time so important? The previous paragraph shows how the TDP can be used to find an efficient multi-goal plan for a specific mission assigned to a mobile robot. Here we explain that such a use case has certain specifics that need to be respected when designing a solver for the TDP. The most crucial aspect is time. In real scenarios, the current instance of the problem continuously changes because, e.g., new sensor data are received, the environment changes unexpectedly, or execution failure occurs. Robots are therefore continually planning and replanning (Ceallaigh and Ruml, 2015). When replanning, a long time spent on recovery (finding a new solution) is unfavorable. Even if we consider a static environment, we still want the robot to start moving (executing the plan) as soon as possible since the time spent finding the solution is added to the total latency, thus the plan depreciates. Lastly, the TDP can be used to solve more complex variants of the MRS, e.g., continuous MRS or a variant where the environment is only partially known (Kulich et al., 2014, 2017). In such a case, frequent replanning can be at the core of a more complex planning framework, and thus rapidly solving a single instance of the TDP becomes crucial. Given the above, the time for obtaining a single TDP solution is usually limited to units of seconds (tens if we are generous). At the same time, naturally, *the best possible quality* of the solution is promoted. Since we deal with an NP-hard problem and the computational time is extremely restricted, it would be unwise to hope for optimal solutions for bigger instances. Therefore, by *the best possible quality*, we mean the best quality that any of the available solvers can provide (within the time limit).

Motivation. The primary motivation of this work is to solve the TDP in the specific context of mobile robotics introduced in Kulich et al. (2014, 2017) and depicted above. In the considered context, the computational time to solve the problem is often limited, or the solution's usefulness depends on the time needed to obtain it. The authors of related works usually do not consider any of these restrictions. The literature seems to follow two main streams. Either the authors seek an exact algorithm that will solve the problem to optimality (e.g., Fischetti et al., 1993; Abeledo et al., 2013; Bulhões et al., 2018), or their approach relies on metaheuristics that are able to find good quality solutions in *reasonable* computing time (e.g., Salehipour et al., 2011; Silva et al., 2012; Mladenović et al., 2013). However, the term *reasonable* is often not well-specified. Usually, it merely holds that — the faster method, the better — as long as its average solution quality is comparable to the current state of the art. Nevertheless, this vague metric is not sufficient to tell which algorithm presented in the literature will provide the best solution after it has run for t_{max} seconds. Therefore, we do not know if existing methods would be sufficiently good in mobile robotics or any context where the computational time is strictly limited. This situation is resolved in our paper. We systematically design a new method for the TDP and experimentally compare it to the current state of the art using various metrics.

Contribution and main contents. In this paper, we present a new metaheuristic for the TDP whose design is based on empirical testing according to a general *run-time distribution* (RTD) methodology (Feo et al., 1994). We motivate the choice of the methodology and explain its specific usage in detail. We also describe in a step-by-step fashion the process of designing the method from various metaheuristic schemes and their parametrizations that we initially considered. Finally, we perform an extensive computational evaluation of our method and compare it with a state-of-the-art reference. Apart from other authors, we present three types of results. We show how the methods behave when the computational time is strictly limited to 1, 2, 5, 10, 20, 50, and 100 seconds. The choice of the time limits

reflects the requirements of mobile robotics presented in previous paragraphs. We also compare the methods by means of *time-to-target* (TTT) plots, which is the most general way of comparison (Resende and Ribeiro, 2016). The TTT-plots can produce the probability that one method finds a solution (of a certain quality) more quickly than the other method. Our third type of result is the one that authors of related works usually provide — mean solution costs and computational times from several runs with a fixed number of iterations.

Structure. The rest of this paper is organized as follows. Related works and the methodology are reviewed in Sec. 2. The TDP is formally defined in Sec. 3. The solution approach, which includes considered algorithms, extended methodology, and description of the proposed metaheuristic, is described in Sec. 4. The proposed metaheuristic is evaluated in Sec. 5. Finally, Sec. 6 is devoted to concluding remarks.

2 Related literature review

Solving the TDP follows two major courses in the operations research community. The first one aims to find the optimal solutions using exact algorithms; however, it is limited to small instances due to infeasible computing times. The second major course seeks just high-quality solutions in exchange for much lower computational times and is applicable even to big instances. Here the core solution methods are heuristics and more general search strategies called metaheuristics. Approximation algorithms, which lie somewhere in the middle, are also known for the TDP. These methods give approximate solutions but, unlike heuristics, with a theoretically proven guarantee of performance. In this branch, the researchers focus primarily on lowering the approximation factor or computational complexity of their algorithms; however, computational results on benchmark instances are usually not present in their works. For the use in mobile robotics, we choose metaheuristics. The choice is natural since optimality is not required, and low computational times are the priority. For completeness, we review all classes of algorithms for the TDP in Subsec. 2.1. Then we review the general methodology suggested by Hoos and Stützle (1998) for evaluating stochastic algorithms in various scenarios in Subsec. 2.2.

2.1 Existing algorithms for the TDP

Early exact algorithms proposed by Lucena (1990); Bianco et al. (1993) rely on non-linear integer formulations in which a *Lagrangian relaxation* is used to derive lower bounds. Fischetti et al. (1993) develop an *integer linear programming* (ILP) formulation and new theoretical results on the matroidal structure of a class of combinatorial problems. The results are used to derive lower bounds for the TDP and are embedded into an enumerative algorithm capable of solving 60-vertices instances to optimality. Other ILP formulations and exact algorithms are proposed in Méndez-Díaz et al. (2008); Ban et al. (2013); Naeni and Salehipour (2019).

In addition to TDP-specialized solutions, several exact approaches are developed for the *time-dependent traveling salesperson problem* (TDTSP), a generalization of both the TSP and the TDP. Some of these formulations are proposed in Gouveia and Voß (1995); Abeledo et al. (2010, 2013); Miranda-Bront et al. (2014); Godinho et al. (2014). Among these, the strongest algorithm is the *branch-cut-&-price* developed by Abeledo et al. (2010, 2013), capable of solving almost all instances from the TSPLIB (Reinelt, 1991) with up

to 107 vertices within the limit of 48 hours. Some larger TSPLIB instances, with up to 150 vertices, are solved by Roberti and Mingozzi (2014) who introduce a new way of computing lower bounds for the TDP based on dynamic *ng*-path relaxation. The last approach is further advanced in Bulhões et al. (2018). Their *branch-&-price* algorithm with improved usage of *ng*-paths, now considered the new state-of-the-art exact algorithm for the TDP, provides new optimal solutions for 13 TSPLIB instances, the largest of which has 195 vertices.

Approximation algorithms for the TDP on a tree and on a general metric graph are developed, e.g., in Blum et al. (1994); Ausiello et al. (2000); Archer and Williamson (2003); Fakcharoenphol et al. (2007); Archer et al. (2008); Archer and Blasiak (2010). Here, solving the TDP *on a graph* means that the particular instance of the problem was constructed using the shortest-path metric from a given graph with arbitrary costs on its edges. The lowest approximation factors in the literature are 3 (Frederickson and Wittman, 2012) and 3.59 (Chaudhuri et al., 2003) for the tree and the general graph respectively.

The heuristic approach mostly relies on general search strategies, especially *variable neighborhood search* (VNS) and *greedy randomized adaptive search procedure* (GRASP). VNS proposed originally by Mladenović and Hansen (1997) is a single-start stochastic metaheuristic based on the idea of improving a single solution by some temporal non-improving steps. In its scheme, two phases alternate: a *shake* which allows escaping local optima and a *local search* phase, which descents towards one. Additionally, a systematic change of neighborhoods within the search is applied. *General VNS* (GVNS) is a variant that uses *variable neighborhood descent* (VND) in the *local search* phase. VND can be seen as a deterministic variant of VNS, which explores a solution space using several neighborhood structures, usually in sequential order. *Greedy randomized adaptive search procedure* (GRASP), unlike VNS, is a multi-start process developed and established within the research community by many authors' works, e.g., Hart and Shogan (1987); Feo and Resende (1989); Feo et al. (1994). *Greedy randomized adaptive* construction heuristic is applied to each restart to create a new solution, which is then improved by VND, and the best overall solution is returned at the end. GVNS and GRASP have similarities and also significant differences. They both use VND as a *local search* method, and both are stochastic to be able to escape local optima — but in a different way. While GVNS randomly *perturbates* the best current solution (in the *shaking* phase), GRASP creates an entirely new one in a randomized fashion and starts the search from the beginning.

Salehipour et al. (2011) propose a GRASP for the TDP that embeds either VND or VNS and evaluate both variants on a set of randomly generated benchmark instances of sizes ranging from 10 to 1000. Silva et al. (2012) later present a simple and effective metaheuristic called GILS-RVND, which is based on the combination of GRASP, *iterated local search* (ILS) with randomized *perturbations*, and *randomized VND* (RVND). It improves all the results obtained by Salehipour et al. (2011) on their instances and finds new best solutions for two of TSPLIB (Reinelt, 1991) instances. Mladenović et al. (2013) propose a GVNS, able to improve the previous results obtained by Salehipour et al. (2011) as well; however, GILS-RVND still performs slightly better in terms of solution quality. Ban et al. (2013) suggest a metaheuristic algorithm combining between *tabu search* (TS) and VNS that uses memory structures to discourage the search from coming to unpromising solutions. The authors show that it compares well with the state-of-the-art algorithms (Salehipour et al., 2011; Silva et al., 2012) in the quality of obtained solutions. The TS-VNS, however, does not improve the results by GILS-RVND in the matter of computional time.

To the best of our knowledge, since its publication in 2012 to this day, the GILS-RVND by Silva et al. has been the one heuristic method providing the best trade-off between its simplicity, solution quality, and computational time in the literature. Thanks to its advantageous

characteristics, it was more recently chosen by other authors as the base method for their improvement ideas. Rios (2016) propose versions of GILS-RVND for parallel computing in CPU/GPU hybrid systems. Santana et al. (2020) improve GILS-RVND by means of *data mining* (DM) techniques. Their new hybrid method, called *multi-DM GILS-RVND* (MDM-GILS-RVND), utilizes the *frequent itemset mining* (FIM) technique to gather segments of high-quality solutions in the first half of GILS-RVND iterations. In the other half, the segments are used to construct new initial solutions with every other restart. MDM-GILS-RVND is shown to perform almost equally as GILS-RVND on small instances ($n \leq 50$), better in terms of computational time on medium instances ($50 < n \leq 200$), and better in both terms of time and solution quality on large instances ($200 < n$).

2.2 Evaluating stochastic algorithms

Hoos and Stützle (1998) point out pitfalls related to stochastic methods evaluation and introduce a methodology for evaluating a certain class of algorithms called *Las Vegas algorithms*. An algorithm $\mathscr{A}$ is said to be a *Las Vegas algorithm* for problem class Π, if (i) whenever for a given problem instance $\pi \in \Pi$ it returns a solution, it is guaranteed to be a valid, and (ii) on each given instance the run-time of $\mathscr{A}$ is a random variable. Hoos and Stützle classify three types of possible application scenarios for *Las Vegas algorithm* $\mathscr{A}$:

1. there are no time limits, i.e., we can afford to run the algorithm as long as it needs to find a valid (or of sufficient quality) solution;
2. there is a time limit t_{max}, which can be very small in case of real-time applications such as robotics;
3. the utility $U : \mathbb{R} \rightarrow [0,1]$ of a solution depends on the time t needed to find it.

It is apparent that evaluating the performance of $\mathscr{A}$ in these scenarios must be done using different criteria for each. E.g., in the case of Type 1, the mean time of several runs might suffice to characterize the run-time (rt) behavior roughly, but it is basically meaningless for Type 2, which needs more adequate criteria such as $P(rt \leq t_{max})$ — the probability of finding a solution within the given time-limit. Also, we can observe that Type 1 and 2 are special cases of the most general Type 3, which can only be appropriately characterized by the *run-time distribution* (RTD) function $rtd(t) = P(rt \leq t)$ or its approximation. In addition, from RTD, other criteria, like the mean run-time, its standard deviation, median, percentiles, or success probabilities $P(rt \leq t_i)$ for arbitrary time-limits t_i, can be extracted. The RTD was first used by Feo et al. (1994) and further addressed by other authors, e.g., Hoos and Stützle (1998), and Aiex et al. (2002). Hoos and Stützle encourage to use the RTD to characterize the behavior of algorithm $\mathscr{A}$ completely and uniquely and stress out the possible pitfalls (such as imprecisions or erroneous conclusions) when other simpler methodologies are used.

As we mentioned earlier, Hoos and Stützle originally proposed the methodology for *Las Vegas algorithms* that either return a valid solution in a finite time rt or do not find any (then $rt = \infty$). However, in the case of TDP, the usual solution methods are improving strategies, i.e., they construct a valid solution in the early stage of their run-time and spend the rest of the time improving it without the loss of validity. In order to apply the RTD to improving strategies a solution cost goal c_{goal} needs to be considered. The function $rtd(t) = P(rt \leq t)$ is then seen as the probability that the algorithm finds a solution with cost $c \leq c_{goal}$ in time $rt \leq t$. The use of this technique is recommended in Resende and Ribeiro (2016) for many problems (including TSP) and the value of c_{goal} is often chosen to be 1% worse than the currently best known solution by the authors.

To conclude, the single most useful (thanks to its universality) way to characterize the run-time of stochastic solution methods for the TDP, which is relevant even in the mobile robotics context, seems to exist. The problem is that it is barely used in the related literature. Instead, other authors use the best and mean solution costs and average values of CPU times, which are higher by far (especially for larger instances) from the time limits imposed by our context (order of seconds). Since neither $rtd(t)$ nor $P(rt \leq t_{max})$ is used to characterize one's algorithm, we can conclude that their results can be relevant only to the solution scenario of Type 1, i.e., limit-less computation times, while for the other two no valid conclusions can be made. Unlike the other authors, in this work, we focus on designing a universally well-performing method in all scenarios 1-3 and use the RTD methodology to achieve it.

3 Problem definition

The *traveling delivery person problem*, also known as the *repairperson problem*, or the *minimum latency problem*, is formally described by: (i) $G = (V, E)$: a complete undirected graph with N vertices $V = \{v_1, \ldots, v_N\}$ in which every pair of distinct ones $v_i \neq v_j$ is connected by a unique edge $e_{i,j} = (v_i, v_j) \in E$; (ii) $d : E \to \mathbb{R}_0^+$: a non-negative cost $d(i, j)$ associated with each edge $e_{i,j}$ representing a length of the shortest path (or travel time) from v_i to v_j; (iii) $s \in V$: a starting vertex (depot) of the delivery person (all other vertices represent the customers). Let the sequence of vertices $\boldsymbol{x} = (x_0 = s, x_1, \ldots, x_n)$, where $n = N - 1$ is the number of customers, be a Hamiltonian path in G starting from the depot. Furthermore, let $d(x_i, x_j)$ be the cost of an edge between i-th and j-th vertex in $\boldsymbol{x}$. The cumulative cost, also denoted as the latency, to reach the k-th vertex in sequence $\boldsymbol{x}$ is defined as

$$\delta_k^{\boldsymbol{x}} = \sum_{i=1}^{k} d(x_{i-1}, x_i) \,. \tag{1}$$

Finally, the total cost of $\boldsymbol{x}$, also denoted as the total latency, is defined as

$$\texttt{Cost}(\boldsymbol{x}) = \sum_{k=1}^{n} \delta_k^{\boldsymbol{x}} = \sum_{k=1}^{n} \sum_{i=1}^{k} d(x_{i-1}, x_i) \,. \tag{2}$$

The objective of the TDP is to find an optimal path $\boldsymbol{x}^\star$ that minimizes the cost, i.e., $\boldsymbol{x}^\star = \arg\min_{\boldsymbol{x} \in \mathscr{H}(\pi)} \texttt{Cost}(\boldsymbol{x})$, where $\pi = (G, d, s)$ is an instance of the TDP and $\mathscr{H}(\pi)$ is the set of all Hamiltonian paths in graph G starting in s.

Note that we regard an open variant of the problem, i.e., travel from the last customer back to the depot is not considered. In the literature, authors sometimes consider a closed variant where a Hamiltonian circuit is considered instead of the path. However, we believe that in the TDP's customer-oriented view, the open variant is a more natural way of defining the problem. A delivery person whose only goal is to satisfy customers (minimize the total latency) would see no benefit in treating the depot as a last imaginary customer regardless if he or she wants to return to the depot after the last delivery. This is different in the TSP's server-oriented view because traveling from the last customer back to the depot clearly adds to the total travel time, which is being minimized. Furthermore, similar arguments apply regarding the MRS problem from mobile robotics shown in Fig. 1. Here, the goal is to search for something in a way that is the most efficient on average. However, while executing the search plan, the actual mission is accomplished once the object is found, and then the robot can abort the rest of the plan. In this case, it is apparent that considering the travel from the last place of the plan to the robot's initial position is meaningless.

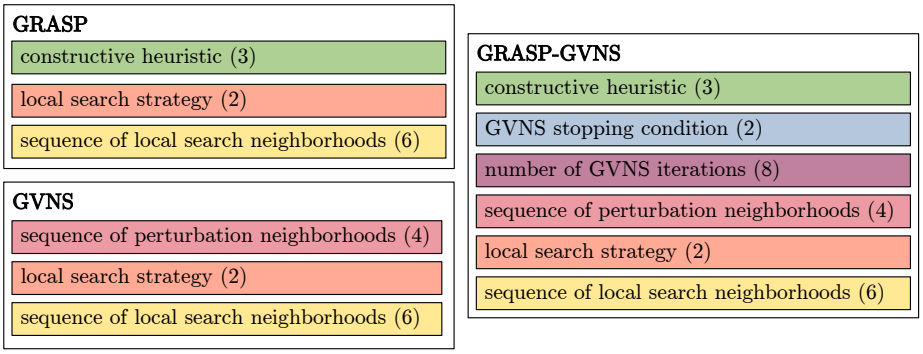


Fig. 2: Considered metaheuristics and their variable components (the number of options is shown in parentheses).

4 Solution approach

This section details the proposed method for the TDP and the process of its design. The process is based on empirical testing of a wide range of algorithms, among which the best one is selected. We consider three general metaheuristic schemes: 1. GVNS, 2. GRASP, 3. a combination of the two, i.e., GRASP that in *local search* uses GVNS instead of VND; and many variants of their subprocedures and parametrizations. The whole scope is outlined in Fig. 2 and detailed in Subsec. 4.1-4.4, and 4.6. We evaluate all combinations according to the RTD methodology, and the overall best one is selected as the proposed method. How the RTD methodology is used is concertized in Subsec. 4.5. Finally, the best method is proposed at the end of Subsec. 4.6. For readers' convenience, we provide a handy overview of symbols that appear in this section in Tab. 1. The union of symbols in the table and symbols defined in Sec. 3 creates a full set of special symbols used in the algorithms' schemes. Any other symbols denote temporal variables created within the schemes.

4.1 General schemes

We consider three general schemes: GVNS, GRASP, and a combination of the two GRASP-GVNS (G+G). The scheme of the hybrid metaheuristic G+G is presented in Alg. 1. The initialization is done on line 2: an iteration counter i is set to 1, a stopping flag *stop* to `false`, and the best solution's cost $c^\star$ to ∞. The main loop (lines 3-23) runs until the *stop* flag is

Sym.	Meaning	Sym.	Meaning
i_{max}	no. of main-loop iter.	s_{rcl}	size of a *restricted candidate list*
c_{goal}	target solution cost	R	set $\{r_i\}$ of $[0,1]$-real values, $i = 1,\ldots,\lvert R\rvert$
t_{max}	CPU time limit	$\boldsymbol{p}$	seq. (p_i) of positive integers, $i = 1,\ldots,\lvert\boldsymbol{p}\rvert$
j_{max}	no. of inner-loop iter.	$\boldsymbol{n}$	seq. $(\mathcal{N}_i)$ of neighborhoods, $i = 1,\ldots,\lvert\boldsymbol{n}\rvert$

Table 1: Overview of special symbols that appear in algorithms.

Algorithm 1: GRASP-GVNS

```
1  Function G+G(i_max, c_goal, t_max, j_max, R, p, n):
2      i ← 1; stop ← false; c* ← ∞
3      while stop = false and i ≤ i_max do          ▷ main G+G loop ~ main GRASP loop
4          α ← random value ∈ R
5          s_rcl ← max(1, ⌊α · N⌋)
6          x ← Construct(s_rcl)
7          j ← 1
8          while stop = false and j ≤ j_max do      ▷ inner G+G loop ~ main GVNS loop
9              k ← 1
10             while stop = false and k ≤ |p| do    ▷ inner GVNS loop
11                 x' ← Shake(x, p_k)
12                 (x', stop) ← Improve(x', t_max, c_goal, n)
13                 if Cost(x') < Cost(x) then
14                     x ← x'
15                     k ← 1
16                     j ← 1                        ▷ only for G+G-b
17                 else
18                     k ← k + 1
19             j ← j + 1
20         if Cost(x) < c* then
21             x* ← x
22             c* ← Cost(x)
23         i ← i + 1
24     return x*
```

`true` or the maximum number of iterations i_{max} is reached. In GRASP, a new solution is constructed in each iteration, then improved and evaluated. The construction (line 6) is done in a greedy randomized fashion where the integer parameter s_{rcl} controls the level of randomness. Admissible values for s_{rcl} are in a range from 1 to N, where N is a size of the instance. 1 corresponds to a purely greedy solution and N to a totally random solution. Parameter s_{rcl} can be either set to some fixed value from the range, or can be constructed as on lines 4-5 of the G+G algorithm. The latter option enables s_{rcl} to vary in each iteration. The GRASP's improvement phase follows on lines 7-19, and the final evaluation is done on lines 20-23. The G+G scheme's improvement phase has embedded the GVNS metaheuristic. To obtain a pure GRASP, lines 7-19 can be replaced by the improvement (VND/RVND) procedure appearing on line 12 with a proper refactorization (replacing $\boldsymbol{x}'$ by $\boldsymbol{x}$).

The embedded GVNS metaheuristic is composed of two additional nested loops — the main GVNS loop (lines 8-19) whose number of interations is controlled by counter j and input parameter j_{max}, and the inner GVNS loop (lines 10-18) controlled by counter k and parameter $|\boldsymbol{p}|$. The parameter $|\boldsymbol{p}|$ is the number of elements (positive integers) of the input sequence $\boldsymbol{p} = (p_1, p_2, \dots)$, which take a role in the *perturbation* phase (line 11), where the k-th member of the sequence is passed to the `Shake` procedure. The procedure is applied to the current solution $\boldsymbol{x}$ and results in a new temporary solution $\boldsymbol{x}'$, which is improved

(line 12) and evaluated (lines 13-18). If the cost of the temporary solution is less than the cost of the current, then the temporary is assigned to the current, and k is reset back to 1. Else, k is incremented, and the loop starts over with the next element of $\boldsymbol{p}$. Note the *stop* flag returned by the improving procedure (line 12). If it is `true`, then the current loop breaks preliminary and the flag propagates to stop the outer loops as well. This way, the whole algorithm can be instantly and safely terminated at any time, returning the incumbent solution $\boldsymbol{x}^\star$, achieving a certain stopping condition modularity discussed in the next Sec. 4.2.

After the whole GVNS subroutine (lines 7-19) ends and returns the current solution $\boldsymbol{x}$, the final evaluation (lines 20-22) finishes the current main GRASP loop iteration. Within one iteration of GRASP, several GVNS iterations are performed. How many depends on the parameter j_{max}. Note line 16, which resets the counter j to 1. It is optional, and it can be either omitted (G+G-a), or not (G+G-b). If omitted, then j_{max} is the exact number of GVNS iterations. If the line is present, then the GVNS iteration counter j resets to 1 with each improvement, and the GVNS loop breaks after j_{max} iterations with no observed improvement. Finally, to obtain a pure GVNS metaheuristic, the following changes must be applied to Alg. 1. In order: j_{max} is replaced by i_{max}, s_{rcl} by 1, $\boldsymbol{x}$ by $\boldsymbol{x}^\star$, and $\boldsymbol{x}'$ by $\boldsymbol{x}$. Then, the following lines are removed: 3-5, 16, and 20-23.

4.2 Stopping conditions

A stopping condition of the general schemes is determined by a tuple of parameters $(i_{max}, c_{goal}, t_{max})$, where i_{max} is the number of main-loop iterations, c_{goal} is the target solution cost, and t_{max} is the CPU time limit. Given the constant i_{max}, the algorithm stops and returns a valid solution after a fixed number of iterations i_{max}, if not stopped earlier by other criteria. Given the CPU time limit, the algorithm finishes, at worst, after t_{max} seconds. At last, the algorithm can also stop after it has found a solution with cost smaller or equal to given goal c_{goal}. The considered algorithms are expected to run in several different modes depending on the combination of stopping conditions. For instance, when i_{max} is some positive integer, $t_{max} = \infty$, and $c_{goal} < 0$, the algorithm will always stop after the fixed number of iterations i_{max}. This configuration is the most common in the literature. Other useful configuration is to set, e.g., $i_{max} = \infty$, and t_{max}, c_{goal} to some reasonable values in accordance with Sec. 4.5. In this case, the algorithm will either stop after it has found a good enough solution or after the time limit has passed. The introduced variability opens a range of different applications and a possibility to generate several types of results used to compare the algorithms in various scenarios.

4.3 Construction and perturbation

The `Construct` procedure implements the *greedy randomized adaptive* construction shown in Alg. 2, lines 1-8. First, a partial solution $\boldsymbol{x}$ is initialized with the depot s and a *candidate list* (CL) with the remaining vertices (line 2). In the main loop (lines 3-7), a *restricted candidate list* (RCL) is built by considering only $\min(s_{rcl}, |\text{CL}|)$ nearest CL elements with respect to the last added vertex to $\boldsymbol{x}$ (line 4). $s_{rcl} \in \{1, \ldots, N\}$ is an argument passed to the procedure and $|\text{CL}|$ is the cardinality of CL. Finally, a candidate is selected from the RCL by random, appended to the end of $\boldsymbol{x}$, and removed from the CL (lines 5-7). The process repeats until the CL becomes empty, i.e., all vertices are added to $\boldsymbol{x}$, and then the finished solution is returned. We consider three variants of the construction procedure in accordance with Fig. 2:

Algorithm 2: G+G subroutines: GRASP construction and GVNS *perturbation*

```
1  Function Construct(s_rcl):
2      x ← s; x ← (x); CL ← V \ {x}
3      while CL is not empty do
4          Create RCL ⊂ CL considering only min(s_rcl, |CL|) nearest candidates to x.
5          x ← random value ∈ RCL
6          Append x to the end of x.
7          CL ← CL \ {x}
8      return x
9  Function Shake(x, p):
10     p ← min(p+1, |x|) − 1                          ▷ adjust p in case |x| < p
11     Create p+1 random subpaths s_0, s_1, ..., s_p of x, where s_0 is the one starting
         with the depot. This can be done by removing p random distinct edges from x.
12     Create sequence of indices i = (1, 2, ..., p) and shuffle it randomly.
13     x' ← s_0
14     foreach i ∈ i do
15         if random Boolean value then
16             Append reversed s_i to the end of x'.
17         else
18             Append s_i to the end of x'.
19     return x'
```

(a) deterministic, (b) randomized with a fixed rate of randomness, (c) randomized with a rate of randomness randomly chosen from a uniform discrete probability distribution. All the variants can be implemented by the `Construct(`s_{rcl}`)` procedure, where (a) $s_{rcl} = 1$, (b) s_{rcl} is fixed in the range from 2 to N, and (c) s_{rcl} is constructed as in lines 4-5 of Alg. 1.

The VNS metaheuristic employs a mechanism that prevents it from getting stuck in local optima. Mladenović and Hansen (1997) call this mechanism *shaking* or the *shake* phase in the original paper where the VNS was introduced. The *shaking* is also present in the VNS's generalized version that we use, and we also refer to it by the term *perturbation*. It resembles the work of Silva et al. (2012), who use the *perturbation* called *double-bridge* for the TDP. *Double-bridge* was originally developed by Martin et al. (1991) for the TSP. It removes and re-inserts four edges from and to the given path such that a new feasible path is generated. The edges to be removed are chosen randomly. Our *perturbation* procedure `Shake` shown in Alg. 2, lines 9-19, generalizes the mechanism by considering p edges instead of four. It works as follows. A valid path $\boldsymbol{x}$ and a positive integer parameter p are passed to the procedure. First, the path is partitioned by removing p random distinct edges (line 11). The subpaths resulting from this operation are labeled as $\boldsymbol{s}_0, \boldsymbol{s}_1, \ldots, \boldsymbol{s}_p$ in the order they appear in $\boldsymbol{x}$. Next, the first subpath $\boldsymbol{s}_0$ is assigned to a partial solution $\boldsymbol{x}'$ (line 13), and a sequence $\boldsymbol{i}$ of indices from 1 to p is created and randomly shuffled (line 12). The algorithm then goes through each index i in the randomized sequence (lines 14-18) and appends either the corresponding $\boldsymbol{s}_i$ (line 18) or its reversed version (line 16) to the end of $\boldsymbol{x}'$. The probability of reversing $\boldsymbol{s}_i$ before appending it to $\boldsymbol{x}'$ is 50%. The partial solution $\boldsymbol{x}'$ becomes feasible after the last remaining subpath is appended, and at this point, the procedure ends and returns $\boldsymbol{x}'$.

Algorithm 3: *Variable neighborhood descent* (VND); *randomized VND* (RVND)

Function `Improve(`$\boldsymbol{x}$, t_{max}, c_{goal}, $\boldsymbol{n}$`):`
  $i \leftarrow 1$; *stop* $\leftarrow$ `false`
  Shuffle sequence $\boldsymbol{n}$ randomly. ▷ only for RVND
  while $i \leq |\boldsymbol{n}|$ **do**
    Denote the i-th neighborhood structure in sequence $\boldsymbol{n}$ as $\mathcal{N}_i$.
    $\boldsymbol{x}' \leftarrow \arg\min_{\tilde{\boldsymbol{x}} \in \mathcal{N}_i(\boldsymbol{x})} \texttt{Cost}(\tilde{\boldsymbol{x}})$
    if `Cost(`$\boldsymbol{x}'$`)` $<$ `Cost(`$\boldsymbol{x}$`)` **then**
      $\boldsymbol{x} \leftarrow \boldsymbol{x}'$
      $i \leftarrow 1$
      if `Cost(`$\boldsymbol{x}$`)` $\leq c_{goal}$ **then**
        stop $\leftarrow$ `true`
        break
      Shuffle sequence $\boldsymbol{n}$ randomly. ▷ only for RVND
    else
      $i \leftarrow i + 1$
    Get the total CPU time t since start.
    if $t \geq t_{max}$ **then**
      stop $\leftarrow$ `true`
      break
  return ($\boldsymbol{x}$, *stop*)

4.4 Local search

The *local search* in all considered general schemes is performed by a method based on *variable neighborhood descent* (VND). VND explores a solution space using several neighborhood structures. Its success relies on the following facts: a local optimum for one neighborhood structure is not necessarily a local optimum with respect to another neighborhood structure, and a global optimum is a local optimum with respect to all considered neighborhood structures. Mjirda et al. (2017) provides an overview of sequential VND variants and their comparison on the TSP. With respect to their notion, we use *basic VND* with *the best improvement strategy*, a variant that performed the best in combination with the GVNS scheme, as the authors report.

Additionally, the order in which the neighborhoods are considered in the VND can be either (a) fixed (deterministic) or (b) randomized. The latter, RVND, randomly selects an available neighborhood to be used in each iteration. Satyananda and Wahyuningsih (2019) compare the performance of VND and RVND on instances of the *capacitated vehicle routing problem*. Here, the selection of operators in random order outperforms the fixed-sequence VND in a matter of solution quality, however, the classical VND usually requires lesser iterations to reach the local optimum. Similar observations report Silva et al. (2012) for the TDP after obtaining some preliminary results. Nevertheless, neither Satyananda and Wahyuningsih nor Silva et al. consider real-time application scenarios as we do. In our context, the supremacy of RVND over VND is not so obvious, especially if the lesser iterations of VND

and complex search strategies (Alg. 1) are considered. Thus, we study both variants in this paper.

The pseudo-code of (R)VND is shown in Alg. 3. The method takes an initial solution as input and returns its improved version in the end. Alternatively, no improvement can be found, and the same solution as the initial is returned. The method also checks if the additional stopping conditions are met. One of them is met, when the total run-time (since the start of the whole algorithm, not just the subprocedure) is over t_{max}. The other one is met, when (R)VND finds a solution with cost $c \leq c_{goal}$. If at least one of these two situations is detected, the method terminates immediately and returns the best solution found so far together with the *stop* flag. The procedure is parametrized by a sequence $\boldsymbol{n} = (\mathcal{N}_1, \mathcal{N}_2, \dots)$ of operators also called neighborhood structures. In general, $\mathcal{N} \in \boldsymbol{n}$ is an operator, which takes a feasible solution $\boldsymbol{x} \in \mathcal{H}(\pi)$ of an instance π and a tuple of parameters as an input and returns a new feasible solution $\boldsymbol{x}' \in \mathcal{H}(\pi)$ as the output. The range of solutions that can possibly be obtained by applying the operator $\mathcal{N}$ on a particular solution $\boldsymbol{x}$ is called the $\mathcal{N}$-neighborhood of $\boldsymbol{x}$ and is denoted as $\mathcal{N}(\boldsymbol{x})$.

The initialization of Alg. 3 is done first. A neighborhood structures counter i is set to 1, the *stop* flag is set to `false` (line 2). In addition, the members of the sequence $\boldsymbol{n}$ are randomly shuffled (line 3) in case of RVND. The main loop follows (lines 4-19). Inside, the best neighbor solution $\boldsymbol{x}'$ of neighborhood $\mathcal{N}_i(\boldsymbol{x})$ is found (line 6). Here, the neighborhood structure $\mathcal{N}_i$ is the i-th in the sequence $\boldsymbol{n}$ and $\boldsymbol{x}$ is the currently best solution. If the cost of $\boldsymbol{x}'$ is less then the cost of $\boldsymbol{x}$ (line 7), then $\boldsymbol{x}'$ is assigned to $\boldsymbol{x}$, counter i is set back to 1 (line 9), and in case of RVND, the sequence $\boldsymbol{n}$ is again shuffled (line 13). In addition, the cost goal check is performed (lines 10-12), with the chance of breaking the main loop and setting the *stop* flag to `true` if the goal-accomplishment condition is satisfied. If the solution improvement condition on line 7 is not satisfied, then the counter i increments by one (line 15), which assures that the next neighborhood structure in the sequence is selected in the next run of the main loop. At last, the run-time goal check is performed (lines 16-18). First, the total CPU time t is obtained (line 16), and then the check is done in analogous way as in case of the cost goal check (lines 17-18). If not terminated prematurely, the main loop finishes after all available neighborhood structures have been tried, and no more improvement was obtained.

The core of the improvement procedure described in previous paragraphs is a systematic exploration of a solution space using several neighborhood structures (operators). We consider operators which are often used for solving TSP, TDP, and other routing problems. The complete set is as follows: `2-opt`, `1-point`, `or-opt2`, `or-opt3`, `or-opt4`, `or-opt5`, `2-point`, and `3-point`. Operator `2-opt` takes two non-adjacent edges from path $\boldsymbol{x}$ and replaces them by two new edges in order to obtain a new feasible path $\boldsymbol{x}'$. All the other operators, unlike `2-opt`, can be defined as a special case of a more general operator that we call `2-string`. The definition of `2-string` is what comes next. Let $\mathcal{Y}$ be a set of all tuples $(\boldsymbol{x}, X, Y, i, j)$ such that $\boldsymbol{x} \in \mathcal{H}(\pi)$, $X \in \{0, 1, \dots, n-1\}$, $Y \in \{\gamma \in \{0, 1, \dots, n-1\} : X + \gamma \leq n\}$, $i \in \{0, 1, \dots, n-X\}$, and $j \in \{\gamma \in \{0, 1, \dots, n-Y\} : \gamma - i \geq X \ \vee \ i - \gamma \geq Y\}$. Then we define operator `2-string` as a relation `2-string` : $\mathcal{Y} \mapsto \mathcal{H}(\pi)$ which takes a string of vertices of size X that come after i-th vertex of $\boldsymbol{x}$ and a string of vertices of size Y that come after j-th vertex of $\boldsymbol{x}$ and interchanges them to create a new path $\boldsymbol{x}' \in \mathcal{H}(\pi)$. With this definition of `2-string`, we can define other operators as the latter with fixed X and Y to some values specific for each operator. For the fixed values of X and Y for all the operators except `2-opt` see Tab. 2.

The computational complexity of exploring the whole neighborhood $\mathcal{N}(\boldsymbol{x})$ for $\mathcal{N} \in \boldsymbol{n}$ (see Alg. 3, line 6) is addressed next. Note that all considered operators take two parameters

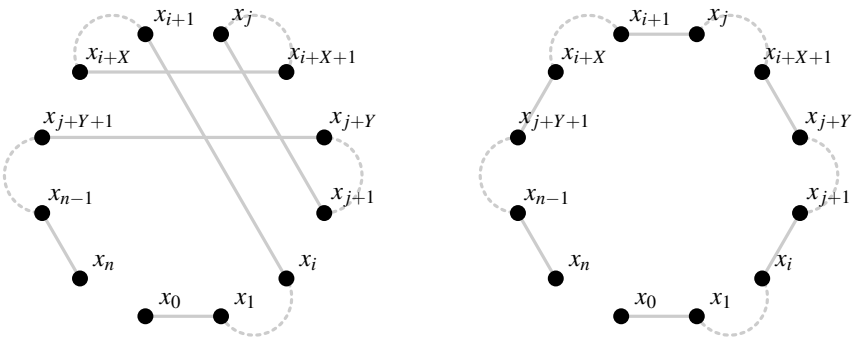


Fig. 3: General operator `2-string(`$\boldsymbol{x}$`, X, Y, i, j)` applied on path $\boldsymbol{x} = (x_k)$ for $k = 0,\ldots,n$, where $0 < X$, $0 < Y$, $i+X < j$, and $j+Y < n$. Edges (x_i,x_{i+1}), (x_{i+X},x_{i+X+1}), (x_j,x_{j+1}), (x_{j+Y},x_{j+Y+1}) are removed and replaced by edges (x_i,x_{j+1}), (x_{j+Y},x_{i+X+1}), (x_j,x_{i+1}), (x_{i+X},x_{j+Y+1}). Left: the original path $\boldsymbol{x}$, right: the resulting path $\boldsymbol{x}'$ obtained by applying the operator on $\boldsymbol{x}$.

Operators:	`1-point`	`or-opt2`	`or-opt3`	`or-opt4`	`or-opt5`	`2-point`	`3-point`
$X =$	0	0	0	0	0	1	1
$Y =$	1	2	3	4	5	1	2

Table 2: All operators except `2-opt` can be defined as `2-string` with fixed X and Y.

(i and j) and all pairs need to be tried when exploring $\mathcal{N}(\boldsymbol{x})$. Therefore, the computational complexity of exploring the whole neighborhood is clearly $\mathcal{O}(n^{2+k})$, where k is the number of additional loops (over the vertices) needed to compute the improvement obtained by applying the operator on $\boldsymbol{x}$ with given parameters i and j. For the TSP, the improvement computation is straightforward without any loop, therefore $k = 0$ and the whole neighborhood can be explored in $\mathcal{O}(n^2)$. Mladenović et al. (2013) show, that the same holds for the TDP, if some additional structures are considered and a pre-processing step is performed. They derive the improvement for `2-opt` and some other operators. For `2-opt`, we use their result. For the general `2-string` operator, we derived the improvement in a similar fashion as shown by the authors.

4.5 Time-to-target plots

This subsection explains how specifically the RTD methodology is used in the design process of our method. More specifically, it introduces a so-called *time-to-target* plot (TTT-plot) that can be used as a metric for comparing algorithms. Consider an instance $\boldsymbol{\pi}$ of an optimization problem class Π, a set of all its valid solutions $\mathcal{H}(\boldsymbol{\pi})$, a cost function $\texttt{Cost} : \mathcal{H}(\boldsymbol{\pi}) \mapsto \mathbb{R}_0^+$, a cost of the optimal solution $c^\star = \min_{\boldsymbol{x}\in\mathcal{H}(\boldsymbol{\pi})} \texttt{Cost}(\boldsymbol{x})$, and a target cost value $c_{goal} \in \mathbb{R}_0^+ : c_{goal} \geq c^\star$. We call algorithm $\mathcal{A}$ a *Las Vegas improving algorithm*, if (i) whenever for a given $\boldsymbol{\pi}$ it returns a solution $\boldsymbol{x}$, it is guaranteed to be valid with

$\texttt{Cost}(\boldsymbol{x}) \leq c_{goal}$, and (ii) for each $\pi \in \Pi$ the run-time rt of $\mathscr{A}$ is a random variable. The algorithm $\mathscr{A}$ runs n_{run} times on the fixed instance π. The runs are assumed to be independent, i.e., the random number generator is initialized with a different seed every time. The rts of all runs are recorded and saved and then used to produce a TTT-plot. TTT-plots construction is well-described in Resende and Ribeiro (2016) and we follow the same methodology shown by the authors. After finishing the last run, the recorded rts are sorted in increasing order and the probability $p_i = (i-1/2)/n_{run}$ is associated with each i-th sorted rt_i, for $i = 1,\ldots,n_{run}$. The meaning of p_i can be understood as follows: p_i is the probability that the algorithm finds a solution at least as good as the target c_{goal} in at most rt_i seconds. Finally, the TTT-plot is constructed by plotting all points (rt_i, p_i). Clearly, the TTT-plot is an approximation of the cumulative RTD capable of characterizing the run-time behavior of *Las Vegas improving algorithms* as we defined them.

Whenever two algorithms $\mathscr{A}_1$ and $\mathscr{A}_2$ are evaluated on the same instance and their TTT-plots are superimposed, it might not be clear on the first sight, which algorithm performs better and by how much. To compare the algorithms productively, some adequate metric must be introduced. Let RT_1 and RT_2 be random variables representing the time needed by algorithms $\mathscr{A}_1$ and $\mathscr{A}_2$ respectively to find a solution as good as the given target value. Let $p_{12} = P(RT_1 \leq RT_2)$ be the probability, that random variable RT_1 takes a value smaller or equal to the value taken by RT_2. Assuming that both algorithms stop when (and only if) they find a solution at least as good as the target, we can say that $\mathscr{A}_1$ performs better than $\mathscr{A}_2$ if $p_{12} > 0.5$. An iterative procedure to compute p_{12} with arbitrary small approximation error for two algorithms is introduced in Ribeiro et al. (2009). Ribeiro and Rosseti (2015) develop a program to compute the approximation of p_{12} from provided TTT-plots of two algorithms. In this work, we use a computation inspired by their program.

4.6 Ms-GVNS: the proposed metaheuristic

This subsection proposes the final metaheuristic and its parametrization, selected as the best among many considered variants in trial testing. All variants are tested on 15 instances of the TDP from Salehipour et al. (2011): TRP-S{50,100,200}-R{1,2,3,4,5}. We choose the best method as the one maximizing a value of a custom-designed metric based on general RTD methodology. The metric is described next. For each pair consisting of a method and an instance, $n_{run} = 200$ runs are performed to compute a single TTT-plot. Each instance's target solution cost is chosen as the 1.01-multiple of the best solution reported by Silva et al. (2012). The computed plot is then superimposed with a TTT-plot of the reference method, GILS-RVND, computed earlier on the same instance, and the probability $p_{tr} = P(RT_{tested} \leq RT_{reference})$ is determined as described in Sec. 4.5. The final metric value for a given method is then computed as a weighted average of p_{tr}'s over the 15 instances, where the weights are sizes of instances (i.e., 50, 100, or 200). This way the metric emphasizes good performance on larger instances. Using the metric, we design the best metaheuristic in two phases. First, we empirically select several promising sets of neighborhoods for the improvement procedure. Second, we combine the promising sets with other parameters and test many configurations. Finally, we select the best-performing method and propose it.

During the first phase, we consider four general schemes: pure GRASP, pure GVNS, G+G-a, and G+G-b; with fixed parametrizations, except the *local search* operators. The two variants of G+G differ in the exclusion / inclusion of line 16 in Alg. 1, respectively. The parameters are fixed to the following values that appear reasonable to us: $\boldsymbol{p} = (4,8,12,16)$, $R = \{.00,.01,\ldots,.25\}$, $j_{max} = 10$. Also, here we prefer RVND over the fixed-sequence

VND. Thus, we do not take the order of neighborhoods into account, and the notion of sets instead of sequences is applicable. For eight considered neighborhoods (presented in Subsec. 4.4), we get $8+28+56+70+56+28+8+1=255$ possible operators' combinations. To reduce the number, we empirically select just some of them. First, a set of all eight neighborhoods $\mathscr{M}_8 = \{\mathscr{N}_{op} : op \in \boldsymbol{ops8}\}$, $\boldsymbol{ops8} = \{$`2-opt`, `1-point`, `or-opt2`, `or-opt3`, `or-opt4`, `or-opt5`, `2-point`, `3-point` $\}$, and its eight corresponding subsets $\mathscr{M}_{8\setminus op} = \mathscr{M}_8 \setminus \{\mathscr{N}_{op}\}$, each containing one lesser neighborhood than $\mathscr{M}_8$, are tested as part of the four considered schemes. The results are averaged over the algorithms and the best set among $\mathscr{M}_{8\setminus op}$, $op \in \boldsymbol{ops8}$, is chosen and denoted as $\mathscr{M}_7$. Then, the set of considered operators is changed to $\boldsymbol{ops7} \leftarrow \boldsymbol{ops8} \setminus (\mathscr{M}_8 - \mathscr{M}_7)$. In the next iteration, sets $\mathscr{M}_{7\setminus op}$, $op \in \boldsymbol{ops7}$, are tested and evaluated analogously as above. The same is repeated for $\mathscr{M}_6, \ldots, \mathscr{M}_1$. Following this strategy strictly, we would obtain 37 sets of operators. However, if the second-best set performs almost equally as the best one, we expand the search from it as well. Using this relaxed strategy, we eventually obtain 57 combinations. Among these, we choose the best six as the most promising. They share some common features. For example, all of them contain operators `2-opt`, `1-point`, and at least one of `or-opt2`, `or-opt3`, or `or-opt4`, and none of them contains `or-opt5`, or `3-point`.

The second phase aims to find the best combination of the general scheme and all of its parameters. The considered schemes are the same as in the previous phase. The considered parameters include the six promising operators' sets; four different configurations of the *perturbation*: $\boldsymbol{p}_1 = (4)$, $\boldsymbol{p}_2 = (4,8)$, $\boldsymbol{p}_3 = (4,8,12)$, and $\boldsymbol{p}_4 = (4,8,12,16)$; and eight different values of the inner iteration constant $j_{max} \in \{10,20,30,40,50,100,150,200\}$. Also, some versions use fixed-sequence VND and some RVND. The constructive heuristic has several different variants as well: fixed $s_{rcl} = 1$ for a deterministic strategy, $s_{rcl} = 3$ for a fixed-randomness strategy, and s_{rcl} constructed as in line 5-6 of Alg. 1 with $R = \{.00, .01, \ldots, .25\}$ for the variable-randomness strategy. Overall, we evaluate more than 2300 methods, among which the best configurations carry some common features. They follow the G+G scheme, they are fully deterministic except the perturbation, and the *perturbation* is parametrized by either $\boldsymbol{p}_3$ or $\boldsymbol{p}_4$.

Ultimately, the best configuration, and the method that we propose, is G+G-b with deterministic construction $s_{rcl} = 1$, *perturbation* parametrized by $\boldsymbol{p} = (4,8,12)$, fixed-sequence VND as the local search strategy, and sequence of neighborhoods $\boldsymbol{n} = (\mathscr{N}_{\texttt{2-opt}}, \mathscr{N}_{\texttt{1-point}}, \mathscr{N}_{\texttt{or-opt2}}, \mathscr{N}_{\texttt{or-opt3}}, \mathscr{N}_{\texttt{or-opt4}})$. The number of GVNS iterations of the best version was originally fixed to $j_{max} = 30$; however, we later found out the method works even better when the value scales with the size of the solved instance, therefore we propose $j_{max} = \lceil size(i)/5 \rceil$. Although the final method is based on the general G+G scheme, its construction heuristic is deterministic, and therefore the relation with GRASP is no longer accurate. Thus, we call the metaheuristic more precisely as *multi-start GVNS* (Ms-GVNS).

An extended description of the method's design, the sub-results that guided us through the design process, and some extra insights are available in Mikula (2021), Appx. A.

5 Computational evaluation

Thorough computational evaluation of Ms-GVNS and its comparison with the reference method follows. As our reference — the state-of-the-art method for the TDP — we choose the original GILS-RVND (Silva et al., 2012) for its simplicity and the right trade-off between solution quality and run-time. We purposely leave out the parallel versions proposed by Rios (2016) since we assume single-processor computing. We also decide not to con-

Inst	%mG	GILS-RVND							Ms-GVNS						
	UB	t_{max} : 1	2	5	10	20	50	100	1	2	5	10	20	50	100
R1	18.31	1.03	0.65	0.31	0.16	0.08	0.03	0.00	0.83	0.60	0.24	0.12	0.05	0.01	0.01
R2	13.15	1.24	1.02	0.59	0.35	0.24	0.07	0.05	0.93	0.71	0.48	0.31	0.20	0.11	0.06
R3	20.61	1.08	0.54	0.29	0.15	0.08	0.02	0.01	0.69	0.35	0.13	0.10	0.06	0.03	0.01
R4	12.65	1.12	0.74	0.30	0.08	0.04	0.01	0.00	0.71	0.41	0.08	0.03	0.01	0.00	0.00
R5	14.85	1.26	0.87	0.46	0.27	0.18	0.07	0.02	0.81	0.53	0.34	0.30	0.15	0.07	0.04
R6	13.28	1.46	0.94	0.41	0.13	0.05	0.00	0.00	1.33	0.70	0.23	0.11	0.03	0.00	0.00
R7	18.39	1.49	1.17	0.53	0.23	0.08	0.01	0.00	1.17	0.70	0.42	0.19	0.09	0.03	0.00
R8	16.60	1.15	0.85	0.29	0.12	0.01	0.00	0.00	0.90	0.54	0.22	0.13	0.05	0.01	0.00
R9	16.94	1.17	0.74	0.46	0.25	0.06	0.00	0.00	1.03	0.78	0.39	0.30	0.09	0.00	0.00
R10	18.03	1.03	0.99	0.44	0.17	0.06	0.03	0.00	1.09	0.70	0.21	0.10	0.05	0.00	0.00
R11	11.49	0.81	0.50	0.24	0.12	0.08	0.02	0.01	0.38	0.29	0.13	0.06	0.03	0.01	0.00
R12	17.78	1.06	0.61	0.34	0.15	0.09	0.03	0.01	0.61	0.31	0.19	0.09	0.05	0.02	0.00
R13	16.66	1.13	0.79	0.35	0.23	0.06	0.02	0.00	0.43	0.23	0.07	0.03	0.01	0.00	0.00
R14	17.04	1.52	0.81	0.43	0.16	0.08	0.01	0.00	1.12	0.54	0.32	0.15	0.05	0.01	0.00
R15	12.69	1.17	0.83	0.40	0.13	0.05	0.00	0.00	0.74	0.51	0.24	0.12	0.04	0.00	0.00
R16	16.96	1.84	1.04	0.68	0.30	0.14	0.02	0.00	1.55	1.05	0.67	0.37	0.19	0.06	0.03
R17	20.35	0.86	0.61	0.32	0.17	0.06	0.02	0.01	0.79	0.61	0.41	0.21	0.11	0.03	0.01
R18	15.52	1.77	1.02	0.33	0.22	0.09	0.02	0.00	1.24	0.69	0.39	0.26	0.22	0.11	0.07
R19	13.22	1.31	0.95	0.45	0.34	0.16	0.05	0.03	0.97	0.71	0.43	0.18	0.12	0.04	0.00
R20	35.21	1.26	0.77	0.25	0.11	0.01	0.00	0.00	0.77	0.43	0.11	0.01	0.00	0.00	0.00
avg	**16.99**	**1.24**	**0.82**	**0.39**	**0.19**	**0.08**	**0.02**	**0.01**	**0.90**	**0.57**	**0.28**	**0.16**	**0.08**	**0.03**	**0.01**

Table 3: *Time-limits* results on 200-customer instances. All times are in seconds.

sider the newest improved version MDM-GILS-RVND (Santana et al., 2020) for reasons we explain next. As the authors' results suggest, the MDM improvement over the original is more significant as the instances go large, e.g., up to 500 or 1000 customers. However, for these instances, the reported run times ($>$ 500 seconds) are still a lot above the range that our paper mainly focuses on ($<$ 100 seconds). Since, in the first half of its iterations, the MDM version is practically identical to its predecessor, it is reasonable to assume that on the instance which takes MDM, e.g., 500 seconds to solve, after 100 seconds of runtime, the average quality of the incumbent solution would be no better than in case of plain GILS-RVND. That being said, we choose the simpler of the two algorithms while obtaining a nearly equivalent comparison with our method as if we have chosen the more complex one. Also, the MDM extension to GILS-RVND introduced by Santana et al. is a general one and might be applied only with minor adjustments to any greedy or semi-greedy restarting heuristic solving any TSP or TDP variant, where the solution is a Hamiltonian path or circuit. As future research, we also consider extending our proposed method to the MDM version. For now, nevertheless, we regard heuristics without DM techniques as they bring no additional value for scenarios we study in this paper.

Both methods, Ms-GVNS and GILS-RVND, are implemented in `C++` under the same framework. All possible parts of the code are shared in order to ensure fairness. This includes improvement computations of all operators as they are described in Sec. 4.4, despite the fact, that Silva et al. use conceptually different improvement calculations for their GILS-RVND. The authors of Silva et al. (2012) were so kind to provide us with their code so we could ensure that our implementation of GILS-RVND is not worse, in any sense, than their implementation. Ms-GVNS (*the proposed*) is parametrized as in Sec. 4.6, and GILS-RVND (*the reference*) as in Silva et al. (2012). All experiments described in this section

Inst	%mG														
	UB	GILS-RVND							Ms-GVNS						
	t_{max} :	1	2	5	10	20	50	100	1	2	5	10	20	50	100
R1	19.1	12.1	6.8	4.8	3.7	2.6	1.6	1.5	6.0	5.2	4.0	3.1	2.2	1.7	1.2
R2	19.9	12.8	7.3	4.5	3.4	2.1	1.4	1.0	5.6	4.5	3.4	2.2	1.5	1.0	0.7
R3	23.8	11.7	6.8	4.8	3.3	2.4	1.5	1.1	5.5	4.5	3.1	2.4	1.7	1.2	0.9
R4	15.9	12.2	7.1	4.8	3.3	2.5	1.7	1.2	5.1	4.2	3.4	2.7	2.2	1.5	1.2
R5	21.7	11.6	6.8	4.7	3.3	2.3	1.7	1.2	5.8	4.7	3.5	2.7	2.2	1.3	1.1
R6	19.9	13.3	6.9	4.4	3.1	2.2	1.5	1.0	5.2	4.2	2.9	2.3	1.5	1.1	0.9
R7	19.9	11.7	7.4	4.8	3.7	2.8	1.7	1.5	5.3	4.7	3.6	2.9	2.2	1.5	1.3
R8	17.2	13.1	7.0	4.8	3.4	2.4	1.6	1.3	5.5	4.9	3.5	2.7	1.9	1.3	0.9
R9	23.8	12.5	7.2	4.5	3.4	2.3	1.5	1.1	4.0	3.5	2.8	2.4	1.8	1.3	1.1
R10	22.8	13.4	7.1	4.9	3.3	2.5	1.5	1.1	3.9	3.1	2.6	2.1	1.6	1.2	0.9
R11	27.3	11.8	6.2	3.8	2.8	1.9	1.2	0.9	4.4	3.6	2.6	1.8	1.4	0.9	0.7
R12	17.3	13.7	7.6	4.9	3.6	2.5	1.6	1.2	5.5	4.9	3.9	2.9	2.3	1.4	1.1
R13	11.6	12.0	6.7	4.7	3.5	2.3	1.7	1.3	4.8	4.0	2.8	2.2	1.6	1.1	0.8
R14	14.2	12.5	7.3	4.5	3.2	2.5	1.6	1.3	4.6	4.1	3.0	2.1	1.8	1.1	1.0
R15	14.1	12.0	7.4	5.0	3.4	2.6	1.7	1.3	3.9	3.2	2.6	1.8	1.3	0.9	0.8
R16	18.7	12.2	6.7	4.3	3.2	2.5	1.6	1.1	5.3	4.2	3.1	2.4	1.8	1.0	0.8
R17	16.4	12.0	6.6	4.3	3.3	2.4	1.6	1.3	5.0	4.6	3.2	2.6	1.7	1.2	0.8
R18	18.3	11.8	6.6	4.6	3.2	2.1	1.3	1.0	5.7	4.8	3.6	2.6	1.9	1.2	0.9
R19	13.1	12.7	7.5	4.7	3.6	2.3	1.4	1.1	5.4	4.6	3.2	2.5	1.7	1.1	0.7
R20	18.4	13.7	7.2	4.7	3.3	2.3	1.6	1.2	5.2	4.5	3.4	2.7	2.2	1.4	1.2
avg	**18.7**	**12.4**	**7.0**	**4.6**	**3.4**	**2.4**	**1.6**	**1.2**	**5.1**	**4.3**	**3.2**	**2.5**	**1.8**	**1.2**	**1.0**

Table 4: *Time-limits* results on 500-customer instances. All times are in seconds.

are executed on a personal computer with Intel® Core™ i7-7700 CPU (3.60 GHz), 32 GB of RAM, and Ubuntu 18.04.1 LTS. The implementation is single-threaded, and only one physical core of the CPU is used for each experiment.

The proposed is tested against *the reference* on several sets of standard benchmark instances generated by Salehipour et al. (2011). The available sets consider 10, 20, 50, 100, 200, 500, and 1000 customers, respectively, and each is composed of 20 random instances. Recall from Sec. 4.6 that instances R1-R5 with 50-200 customers (15 in total) are used in the process of designing *the proposed*. Performance on the remaining 125 instances should demonstrate how general is the method's obtained configuration. However, we should not limit this demonstration only to random instances. Thus, we also consider the ten instances selected by Salehipour et al. from TSPLIB (Reinelt, 1991), sized 70-532. For the evaluation, we use three different metrics that we call *time-limits*, *TTT-plots*, and *fixed-iters*.

1. ***Time-limits, our key results:*** a computational time limit is given to the method, 50 runs are performed, and the solution costs are recorded. This approach is realized on instances with 200, 500, and 1000 customers and those from TSPLIB. The considered time limits are 1, 2, 5, 10, 20, 50, and 100 seconds. Recall from Sec. 1 that we focus on solving the problem under strict time constraints with the motivation to later solve some more complex problems from mobile robotics. Thus *time-limits* are the key results as they reflect this intention. A practical advantage of this result type is that the total time required by the experiments is reasonably bounded regardless of instance size.
2. ***TTT-plots, complementary results (1):*** plots from 200 executions are obtained, and the probability $p_{pr} \approx P(RT_p < RT_r)$ is computed as described in Sec. 4.5. Here, RT_p and RT_r are random variables representing the time needed by *the proposed* and *the reference* re-

Inst	%mG	GILS-RVND							Ms-GVNS						
	UB	t_{max} : 1	2	5	10	20	50	100	1	2	5	10	20	50	100
R1	18.3	676.1	502.6	247.2	93.6	12.3	6.9	5.7	9.4	8.1	6.1	5.6	5.2	4.2	3.5
R2	23.6	679.5	626.6	350.8	132.2	21.2	7.7	5.9	9.8	8.4	6.8	6.2	5.3	4.4	3.8
R3	15.6	685.2	588.8	355.2	117.5	17.9	6.9	5.4	10.7	9.9	7.8	6.6	5.7	4.6	3.8
R4	15.7	673.8	664.0	328.9	133.6	21.0	7.5	5.6	8.7	7.3	6.2	5.4	4.9	4.3	3.7
R5	21.8	766.3	586.0	313.7	142.4	21.5	7.5	5.6	9.3	8.1	6.3	5.8	5.4	4.7	4.2
R6	19.0	629.9	586.6	342.0	136.5	23.9	7.9	5.8	10.9	9.9	7.9	7.0	6.4	5.3	4.6
R7	19.9	720.8	636.7	368.9	140.2	27.3	7.9	6.5	10.9	9.6	7.8	7.3	6.6	5.7	4.9
R8	18.6	760.0	542.4	348.5	126.4	23.4	7.3	5.5	10.5	9.3	7.8	7.0	6.1	4.8	4.0
R9	22.6	675.8	617.0	368.8	126.8	24.5	8.3	6.2	11.7	10.9	8.9	8.1	7.0	5.6	4.5
R10	15.6	761.7	508.9	312.0	160.7	24.9	8.0	6.1	11.1	10.0	8.2	7.6	6.5	5.5	4.6
R11	20.6	688.0	586.8	335.0	137.0	25.9	7.7	6.2	11.5	9.9	7.4	7.0	6.1	4.8	4.0
R12	19.8	687.7	584.5	381.2	153.9	27.2	8.3	6.5	10.5	9.5	7.9	7.0	6.4	5.3	4.4
R13	19.1	656.3	599.2	355.8	143.3	26.4	7.6	6.0	11.3	10.1	8.1	6.6	6.0	4.9	4.1
R14	17.3	683.0	556.5	365.2	145.0	24.8	7.2	5.7	8.4	7.2	6.2	5.4	4.9	4.0	3.5
R15	15.9	674.2	645.3	327.8	143.7	23.6	7.7	5.7	9.2	8.1	6.3	5.7	4.9	4.2	3.5
R16	17.5	692.5	562.0	358.4	138.1	27.8	7.8	5.9	11.3	10.0	8.3	7.3	6.1	4.7	3.9
R17	20.3	661.2	571.6	316.9	144.1	27.0	7.8	5.9	10.2	9.4	8.5	7.4	6.6	5.3	4.4
R18	23.8	670.4	649.2	365.6	134.8	29.0	8.3	6.4	10.8	9.7	7.8	7.1	6.5	5.5	4.5
R19	18.7	737.2	593.3	375.0	141.9	28.7	7.7	5.8	9.5	8.1	6.9	6.3	5.7	4.6	3.8
R20	24.1	776.6	559.6	333.2	142.1	29.1	7.9	6.0	9.2	8.3	7.2	6.6	6.0	4.8	4.1
avg	**19.4**	**697.8**	**588.4**	**342.5**	**136.7**	**24.4**	**7.7**	**5.9**	**10.2**	**9.1**	**7.4**	**6.7**	**5.9**	**4.9**	**4.1**

Table 5: *Time-limits* results on 1000-customer instances. All times are in seconds.

Inst	%mG	GILS-RVND							Ms-GVNS						
	UB	t_{max} : 1	2	5	10	20	50	100	1	2	5	10	20	50	100
st70	12.05	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
rat99	10.97	0.14	0.05	0.01	0.00	0.00	0.00	0.00	0.18	0.11	0.02	0.01	0.00	0.00	0.00
kroD100	13.87	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
lin105	18.19	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
pr107	4.92	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.11	0.10	0.05	0.02	0.00	0.00	0.00
rat195	4.18	1.48	1.15	0.43	0.23	0.10	0.02	0.02	0.11	0.03	0.02	0.02	0.01	0.01	0.00
pr226	56.68	0.11	0.06	0.01	0.00	0.00	0.00	0.00	0.03	0.01	0.00	0.00	0.00	0.00	0.00
lin318	26.23	3.40	2.33	1.16	0.64	0.38	0.27	0.07	1.23	0.88	0.62	0.38	0.25	0.16	0.07
pr439	22.85	6.42	4.46	2.44	1.33	1.06	0.57	0.34	2.56	2.13	1.38	0.86	0.71	0.40	0.16
att532	44.17	16.95	6.27	4.01	2.74	1.79	1.05	0.73	5.14	4.22	3.04	2.31	1.56	0.94	0.67

Table 6: *Time-limits* results on TSPLIB instances. All times are in seconds.

spectively to find a solution that is as good as given c_{goal}. This approach is realized on instances with 10, 20, 50, 100, 200, and 500 customers and those from TSPLIB. The target value c_{goal} is the optimum for instances 10, 20, 50, where it is known (Salehipour et al., 2011; Silva et al., 2012), and the best solution reported by Silva et al. worsened by 1% for instances 100, 200, and 500. For TSPLIB instances, we use c_{goal} obtained as our best-found solution worsened by 1%. We do not use the best solutions reported by other authors for TSPLIB instances because they treat them as a closed TDP with a Hamiltonian *circuit* as the solution. (Recall from Sec. 3 that we consider an open variant

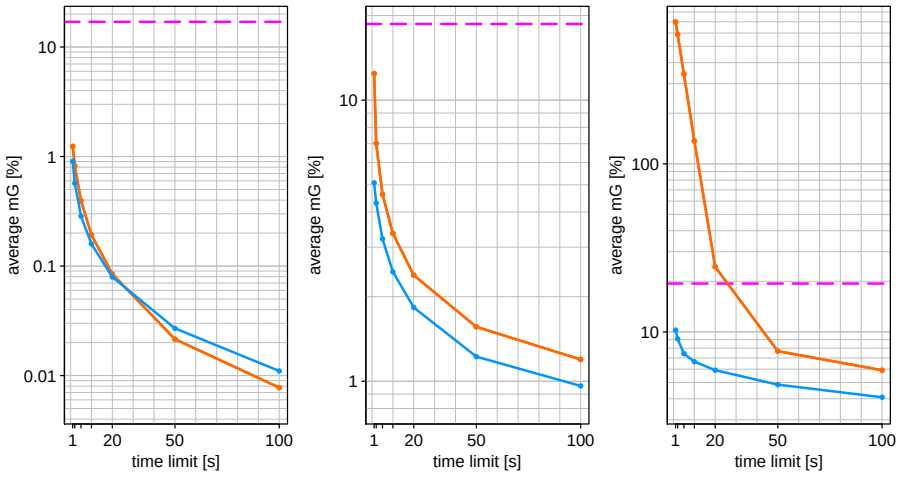
with a Hamiltonian *path* as the solution.) Lastly, since the factor 1.01 used to worsen the best solution is arbitrary in essence, we also test different values: 1.02, 1.05, 1.10, and 1.20 on 500-customer instances and analyze the outcomes.

3. ***Fixed-iters, complementary results (2):*** a fixed number of iterations $i_{max} = 10$ is given to the method, 10 runs are performed, and returned solution costs and run-times are recorded. This approach is realized on instances with 10, 20, 50, 100, 200, and 500 customers and those from TSPLIB. This type of evaluation is the most common in the literature.

We intentionally leave out 1000-customer instances for the complementary results because of the tremendous time requirements of related experiments. Nevertheless, we believe that the remaining results provide a sufficiently detailed view of our method's performance and behavior.

In the tables presented hereafter, c_{best} is the best-known solution, Best and Mean denote the best and the mean solution cost found by the examined method respectively, %bG, and %mG are the best, and a mean percentage gap from c_{best} computed as $100 \cdot (\text{Best} - c_{best})/c_{best}$, and $100 \cdot (\text{Mean} - c_{best})/c_{best}$ respectively, and Time is the average run-time over 10 executions. For instances with up to 200 customers, c_{best} corresponds to values reported by Silva et al. (2012), while for 500 and 1000 -customer instances they are the minimum of values reported by Silva et al. (2012), Rios (2016), and Santana et al. (2020). For TSPLIB instances, c_{best} is the best solution found by *the reference* in our experiments (we do not use other authors' values because they consider the closed variant, similarly as with c_{goal} for *TTT-plots*). In the context of *TTT-plots*, we report TTT, as the average (over 200 runs) time to target solution, and $\%p_{pr}$ as the probability p_{pr} in percents. An over-lined symbol (e.g., $\overline{\%\text{bG}}$) indicates that the value is averaged over instances of the same size. We use blue and orange colors to emphasize %bG, %mG, Time, TTT, and $\%p_{pr}$ values where Ms-GVNS and GILS-RVND respectively performed better than the other. If the same values for both methods are black, then the methods performed equally. If one of the values is in color, but the values appear equal, then the difference was lost after rounding. Additionally, we note that while generating *fixed-iters* results on instances of sizes 10-500, we performed the same experiments as Silva et al. (2012), however with our own implementation, improvement calculations, different random number generator seeds and using more powerful hardware. For %bG and %mG, our obtained results are identical (except some small statistical error), to the ones reported by the authors. On the other hand, our values of Time are significantly lower. Thus, we report unchanged values of %bG, %mG from Silva et al. (2012), and our updated values of Time in the results.

To produce our main results, *time-limits*, we simulate a scenario where the algorithms are allowed to run for a fixed time t_{max}, and then they immediately return a solution whose cost is recorded. Seven different values for t_{max} are used (1, 2, 5, 10, 20, 50, and 100 seconds), and 50 runs are executed for each experimental setup consisting of a method, an instance, and a time-limit. The values of %mG over 50 runs on instances with 200, 500, and 1000 customers and TSPLIB instances are presented in Tables 3, 4, and 5, respectively. The column called %mG of UB shows the mean gaps of upper bounds from the best-known solutions. The upper bounds are obtained by the deterministic greedy algorithm used to generate the initial solution in Ms-GVNS. For 200-customer instances, both Ms-GVNS and GILS-RVND return a better solution than the upper bound even after one second of computing time. As expected, the proposed method converges more quickly towards c_{best}. After 100 seconds, the solution returned by both methods is of similar quality. For 500-customer instances, the values after one second are closer to the upper bound, and after 100 seconds,



(a) 200 customers (b) 500 customers (c) 1000 customers

Fig. 4: Convergence of the average mean gap from Tables 3, 4, and 5 for Ms-GVNS (blue) and GILS-RVND (orange). The dashed magenta horizontal line is the average %mG of the upper bound.

Ms-GVNS typically still returns higher-quality solutions. The convergence of the average %mG can be better tracked in Fig. 4, which shows the last rows (avg) of these tables plotted as graphs. See the graph for the largest instances with 1000 customers. Here, after one second, *the proposed* returns a solution better than the upper bound and converges to the best-known solution. However, *the reference* starts significantly above the upper bound and gets below it typically after no less than 20 seconds. This behavior is caused by generating the initial solution in a randomized way as it is done by GRASP. Such an initial solution can be very far from the optimum, and it takes many operations to get anywhere nearby it. This behavior could be neglected by generating a purely greedy solution at first, saving it as the incumbent, and then run GILS-RVND as usual. Note that this improvement would merely change the behavior of Ms-GVNS, which works with the purely greedy solution from the start, and provides better-than-UB solutions after no more than one second. Even if we considered the improved *reference* as suggested, after 100 seconds Ms-GVNS would still return better solutions which are on average by 1.8% closer to c_{best} than solutions returned by *the reference*. Based on these observations, we claim that Ms-GVNS is superior to GILS-RVND in most scenarios with up to 1000 customers and a hard limit on computational time ranging from 1 to 100 seconds. Additionally, Tab. 6 shows that the same holds for the selected instances from TSPLIB.

The complementary results, *fixed-iters* and *TTT-plots*, are summarized in Tab. 7 for instances with 10, 20, 50, 100, and 200 customers. For instances with up to 50 customers, both methods found the optimal solution in all executions over all instances and sizes, i.e., the average values of %bG and %mG are both exactly zero. For 100 and 200-customer instances, GILS-RVND's %mG is slightly better than Ms-GVNS's, as well as is %bG for 200-customer because Ms-GVNS did not find the best-known solution for two instances (R1 and R2). On the other hand, the Time improvement of Ms-GVNS over GILS-RVND is significant: 37%, 82%, 118%, 136%, and 40% for the considered sizes, respectively. The average *TTT-plots*

	GILS-RVND				Ms-GVNS				
Size	%bG	%mG	Time	TTT	%bG	%mG	Time	TTT	$\%p_{pr}$
11	0.000	0.000	$1.3 \cdot 10^{-3}$	$5.2 \cdot 10^{-5}$	0.000	0.000	$9.0 \cdot 10^{-4}$	$4.5 \cdot 10^{-5}$	61.9
21	0.000	0.000	$1.4 \cdot 10^{-2}$	$7.1 \cdot 10^{-4}$	0.000	0.000	$7.4 \cdot 10^{-3}$	$5.2 \cdot 10^{-4}$	70.5
51	0.000	0.000	$2.6 \cdot 10^{-1}$	$2.5 \cdot 10^{-2}$	0.000	0.000	$1.2 \cdot 10^{-1}$	$1.5 \cdot 10^{-2}$	69.0
101	0.000	0.000	$3.3 \cdot 10^{0}$	$9.8 \cdot 10^{-2}$	0.000	0.002	$1.4 \cdot 10^{0}$	$5.1 \cdot 10^{-2}$	70.9
201	0.000	0.035	$2.9 \cdot 10^{1}$	$2.0 \cdot 10^{0}$	0.004	0.080	$2.1 \cdot 10^{1}$	$1.1 \cdot 10^{0}$	67.7

Table 7: Average *Fixed-iters* and *TTT-plots* results on instances with 10, 20, 50, 100, and 200 customers. All times are in seconds.

			GILS-RVND				Ms-GVNS						
Inst	c_{best}	c_{goal}	%bG	%mG	Time	TTT	Best	%bG	Mean	%mG	Time	TTT	$\%p_{pr}$
R1	1841210	1859799	0.01	0.80	615.3	371.9	1848215	0.38	1852248.7	0.60	952.3	282.4	56.15
R2	1815664	1834733	0.05	0.41	608.8	112.5	**1815478**	−0.01	1818765.5	0.17	806.1	58.0	71.30
R3	1826738	1851374	0.35	0.69	627.0	84.9	1830510	0.21	1833469.3	0.37	939.6	59.9	62.76
R4	1802921	1827358	0.35	0.72	635.7	128.8	1803003	0.00	1811107.7	0.45	934.8	82.4	63.45
R5	1821250	1842214	0.15	0.70	556.8	185.8	1823213	0.11	1829110.9	0.43	837.5	111.7	62.76
R6	1782731	1804486	0.22	0.46	615.4	98.0	1786903	0.23	1790935.4	0.46	806.7	48.0	72.67
R7	1846251	1866478	0.09	0.63	666.6	275.2	1847322	0.06	1853758.1	0.41	936.6	165.2	63.68
R8	1819636	1839054	0.07	0.53	618.9	248.4	**1818621**	−0.06	1826246.4	0.36	893.3	154.9	62.40
R9	1729796	1751157	0.23	0.42	599.0	86.1	1734166	0.25	1739277.9	0.55	794.9	101.1	45.97
R10	1761174	1780368	0.09	0.35	624.1	148.4	1762984	0.10	1766260.3	0.29	874.9	102.8	62.21
R11	1797771	1815859	0.01	0.21	530.6	101.1	**1797111**	−0.04	1801042.4	0.18	825.9	49.0	72.26
R12	1774452	1792196	0.00	0.53	575.4	151.9	1775230	0.04	1780101.6	0.32	948.5	161.8	48.70
R13	1863905	1892435	0.53	0.76	625.2	88.0	1865963	0.11	1870876.1	0.37	817.7	34.2	80.68
R14	1796129	1817162	0.17	0.53	667.7	164.6	1798223	0.12	1802641.2	0.36	888.1	67.4	73.77
R15	1784919	1809056	0.35	0.71	621.4	125.5	1786988	0.12	1791543.4	0.37	780.7	25.2	91.22
R16	1804392	1828289	0.32	0.67	637.6	103.5	1806297	0.11	1809260.2	0.27	879.1	45.6	75.20
R17	1819909	1844005	0.32	0.80	594.0	112.9	1823132	0.18	1825989.3	0.33	926.1	54.4	71.85
R18	1825615	1844525	0.04	0.42	650.1	104.0	1825659	0.00	1829417.4	0.21	904.3	77.1	60.15
R19	1776855	1797040	0.13	0.33	598.1	105.2	**1775030**	−0.10	1779258.1	0.14	911.2	59.9	68.54
R20	1820168	1839021	0.04	0.57	623.4	213.8	1822641	0.14	1828615.0	0.46	942.5	165.4	56.79
avg	-	-	**0.18**	**0.56**	614.6	**150.5**	-	**0.10**	-	**0.36**	**880.0**	**95.3**	**66.13**

Table 8: *Fixed-iters* and *TTT-plots* results on instances with 500 customers. All times are in seconds.

results are favorable towards Ms-GVNS. The average improvement in reaching the target solution is 15%, 38%, 64%, 91%, and 73%, respectively, and the average probabilities of returning the target solution before *the reference* are 62%, 71%, 69%, 71%, and 78% respectively.

Detailed *fixed-iters* and *TTT-plots* results for 500-customer instances are shown in Tab. 8. Here, *the proposed* provides better gaps; however, the run-time of classical *fixed-iters* experiments is worse, on average, by about 43% when compared to *the reference*. The same can be noticed in Tab. 9 for TSPLIB instances. It seems like the roles of the two algorithms switch as we consider large problems. This observation is caused by two factors specific to the *fixed-iters* computational context: (i) the longer the method runs, the better solution usually returns at the end, and (ii) the run-time of the method depends on the number of tries to improve the incumbent solution before it gives up and moves to the next iteration. Regarding the latter, GILS-RVND gives up after $\min(size(i), 100)$, and Ms-GVNS after $\lceil size(i)/5 \rceil \cdot |\boldsymbol{p}|$

			GILS-RVND				Ms-GVNS						
Inst	c_{best}	c_{goal}	%bG	%mG	Time	TTT	Best	%bG	Mean	%mG	Time	TTT	$\%p_{pr}$
st70	19710	19907	0.00	0.00	0.8	0.0	19710	0.00	19710.0	0.00	0.3	0.0	71.52
rat99	56573	57138	0.00	0.00	3.9	0.1	56573	0.00	56718.2	0.26	1.5	0.1	58.34
kroD100	951609	961125	0.00	0.00	2.9	0.1	951609	0.00	951609.0	0.00	1.1	0.0	90.91
lin105	586751	592618	0.00	0.00	2.7	0.0	586751	0.00	586751.0	0.00	1.1	0.0	84.50
pr107	1981991	2001810	0.00	0.00	3.3	0.0	1984642	0.13	1984642.0	0.13	1.4	0.0	97.37
rat195	216154	218315	0.00	0.06	28.9	2.5	216154	0.00	216184.7	0.01	12.5	0.1	92.46
pr226	7101223	7172235	0.00	0.00	23.7	0.2	7101223	0.00	7101223.0	0.00	26.6	0.0	92.58
lin318	5569520	5625215	0.00	0.13	95.2	7.1	5569520	0.00	5572658.6	0.06	96.2	2.2	84.18
pr439	17724562	17879682	0.00	0.30	245.8	28.5	17702656	−0.12	17742375.1	0.10	301.5	16.8	68.52
att532	17449404	17620355	0.00	0.25	707.2	72.1	17445897	−0.02	17481759.1	0.19	1162.6	67.7	53.80
avg	-	-	**0.00**	**0.07**	-	-	-	**0.00**	-	**0.07**	-	-	**79.42**

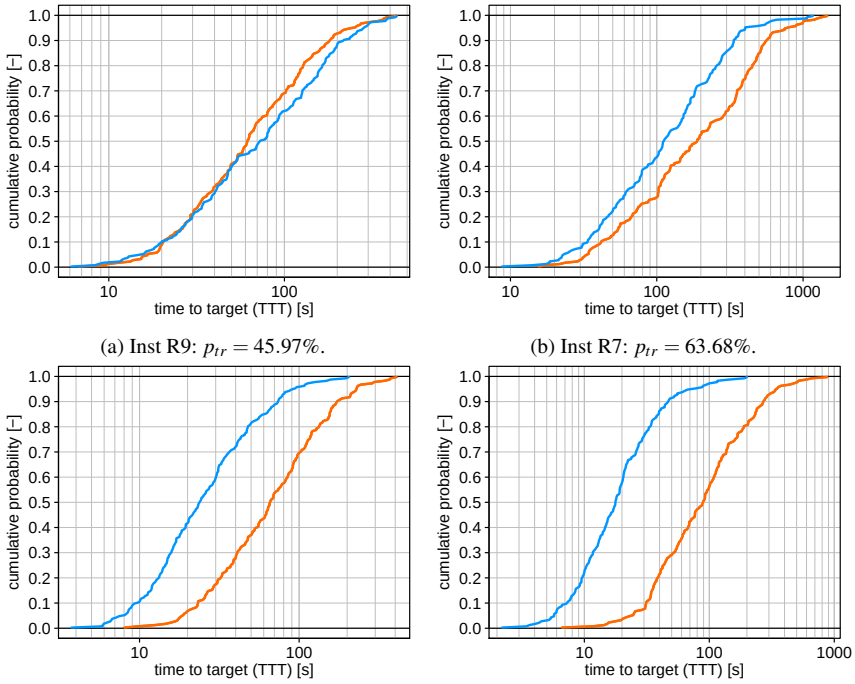
Table 9: *Fixed-iters* and *TTT-plots* results on TSPLIB instances. All times are in seconds.

	c_{goal}	GILS-RVND	Ms-GVNS	
		$\overline{\text{TTT}}$	$\overline{\text{TTT}}$	$\overline{\%p_{pr}}$
1.01 × Silva et al. (2012)		150.5	95.3	66.13
1.02 × Silva et al. (2012)		31.3	18.8	72.98
1.05 × Silva et al. (2012)		4.6	1.3	94.10
1.10 × Silva et al. (2012)		1.2	0.1	98.81
1.20 × Silva et al. (2012)		0.9	0.0	97.15

Table 10: Average *TTT-plots* results with different target solution costs on instances with 500 customers. All times are in seconds.

tries on instance i. In other words, for instances with 100, 200, and 500 customers, *the reference* retains the number of tries fixed to 100, but *the proposed* performs 63, 123, and 303 tries, respectively. Thus, Ms-GVNS is more thorough for larger instances than GILS-RVND, which results in better gaps and longer running times, but for smaller instances, it is the opposite. Thus, based on *fixed-iters* results alone, it is problematic to make an absolute statement about which method performs better in general. Although *fixed-iters* computational context may provide useful, informative results, it is not suitable for the absolute comparison of the methods. Rather, it is more relevant to compare the methods according to their p_{pr} in the *TTT-plots* context, as discussed in Sec. 2.2, and 4.5. In terms of probability p_{pr}, *the proposed* is almost always better than *the reference*, except for two 500-customer instances. The average p_{pr} is about 66% for 500-customer instances and 79% for TSPLIB instances of various sizes. To complement Tab. 8, we show the TTT-plots for four selected instances in Fig. 5: one with the lowest p_{pr} (5a), one with p_{pr} close to median (5b), and two with the highest p_{pr} (5c, 5d). Tab. 8 also shows in **bold** four newly found best-known solutions for instances TRP-S500-R2, TRP-S500-R8, TRP-S500-R11, and TRP-S500-R19.

The last thing that we analyze is the influence of c_{goal} on *TTT-plots*. So far, the factor used to worsen the best solution was 1.01. To provide a complete picture of how the two methods behave, we repeated the experiments whose results are shown in Tab. 8 also for other values of c_{goal}. The summary is shown in Tab. 10. We deduce that the more accessible the goal is, the faster Ms-GVNS finds it compared to GILS-RVND. The average p_{pr} for factors 1.01, 1.02, 1.05, 1.10, and 1.20 is about 66%, 73%, 94%, 99%, and 97%, respectively.



(a) Inst R9: $p_{tr} = 45.97\%$.

(b) Inst R7: $p_{tr} = 63.68\%$.

(c) Inst R13: $p_{tr} = 80.68\%$.

(d) Inst R15: $p_{tr} = 91.22\%$.

Fig. 5: TTT-plots for Ms-GVNS (blue) and GILS-RVND (orange) on four instances with 500 customers.

6 Conclusions

In this paper, we propose a new metaheuristic called Ms-GVNS based on *general variable neighborhood search* with restarts, deterministic *variable neighborhood descent*, and custom *double-bridge* inspired *perturbations* for solving the *traveling delivery person problem*. The method's performance is assessed by three types of experiments: 1. with hard upper limit on computational time (*time-limits*), 2. with target solution cost (*TTT-plots*), and 3. with fixed number of iterations (*fixed-iters*). Experiments of the first type, *time-limits*, are executed on standard benchmark instances with up to 1000 customers, and the proposed method stably outperforms the reference metaheuristic, GILS-RVND, suggested by Silva et al. (2012). Therefore, Ms-GVNS is suitable for real-time applications, e.g., in mobile robotics, where the best possible solution is required while the computational time is bounded by a hard limit. We present this finding as the key result of our paper.

Additionally, we also provide *fixed-iters* and *TTT-plots* results based on experiments over a subset of the benchmark instances. In the literature's most classical context, *fixed-iters*, the proposed method provides better quality solutions in exchange for longer runtimes for instances of size 500. For smaller instances, this trend is observed reversely with the reference method. For the 500-customer instances, the proposed method found four new best-known solutions. Regarding the most general type of evaluation, *TTT-plots*, Ms-GVNS is almost always better than the reference, except for two (out of 20) 500-customer instances. Analysis of different target solution costs has shown that the more accessible the goal is, the faster Ms-GVNS finds it compared to the reference.

As future research, we consider creating a general framework for the *mobile robot search*, where the here-introduced metaheuristic is deployed. In this context, the restrictions on computational time studied in this paper arise. Regarding future improvements of the proposed metaheuristic, we want to extend it to the MDM version similarly to what Santana et al. (2020) did with GILS-RVND. The application of DM techniques will not change the main results presented in this paper. It will, however, further improve the performance of Ms-GVNS on larger instances and in scenarios without strict time constraints.

Acknowledgements

This work has been supported by the European Union's Horizon 2020 research and innovation program under grant agreement No. 688117, the project Rob4Ind4.0 CZ.02.1.01/0.0/0.0/15_003/0000470, and the European Regional Development Fund. The work of Jan Mikula was also supported by the Grant Agency of the Czech Technical University in Prague, grant No. SGS21/185/OHK3/3T/37. The authors would also like to thank Marcos Silva, who kindly provided us his code and datasets.

References

Abeledo H, Fukasawa R, Pessoa A, Uchoa E (2010) The time dependent traveling salesman problem: Polyhedra and branch-cut-and-price algorithm. In: Festa P (ed) Experimental Algorithms, Springer Berlin Heidelberg, Berlin, Heidelberg, pp 202–213

Abeledo H, Fukasawa R, Pessoa A, Uchoa E (2013) The time dependent traveling salesman problem: polyhedra and algorithm. Mathematical Programming Computation 5(1):27–55

Aiex RM, Resende MG, Ribeiro CC (2002) Probability distribution of solution time in GRASP: An experimental investigation. Journal of Heuristics 8(3):343–373

Archer A, Blasiak A (2010) Improved Approximation Algorithms for the Minimum Latency Problem via Prize-Collecting Strolls. In: Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms, Society for Industrial and Applied Mathematics, Philadelphia, PA, pp 429–447

Archer A, Williamson DP (2003) Faster approximation algorithms for the minimum latency problem. In: Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms, SIAM, pp 88–96

Archer A, Levin A, Williamson DP (2008) A Faster, Better Approximation Algorithm for the Minimum Latency Problem. SIAM Journal on Computing 37(5):1472–1498

Ausiello G, Leonardi S, Marchetti-Spaccamela A (2000) On Salesmen, Repairmen, Spiders, and Other Traveling Agents. In: Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), pp 1–16

Ban HB, Nguyen K, Ngo MC, Nguyen DN (2013) An efficient exact algorithm for the Minimum Latency Problem. Progress in Informatics (10):167

Bianco L, Mingozzi A, Ricciardelli S (1993) The traveling salesman problem with cumulative costs. Networks 23(2):81–91

Blum A, Chalasani P, Coppersmith D, Pulleyblank B, Raghavan P, Sudan M (1994) The minimum latency problem. In: Proceedings of the twenty-sixth annual ACM symposium on Theory of computing - STOC '94, ACM Press, New York, New York, USA, pp 163–171

Bulhões T, Sadykov R, Uchoa E (2018) A branch-and-price algorithm for the Minimum Latency Problem. Computers & Operations Research 93:66–78

Campbell AM, Vandenbussche D, Hermann W (2008) Routing for Relief Efforts. Transportation Science 42(2):127–145

Ceallaigh DO, Ruml W (2015) Metareasoning for Concurrent Planning and Execution. Proceedings of the Symposium on Combinatoral Search

Chaudhuri K, Godfrey B, Rao S, Talwar K (2003) Paths, trees, and minimum latency tours. In: 44th Annual IEEE Symposium on Foundations of Computer Science, 2003. Proceedings., IEEE Computer. Soc, pp 36–45

Cook WJ (2012) In Pursuit of the Traveling Salesman: Mathematics at the Limits of Computation. Princeton University Press

Faigl J, Hollinger GA (2014) Unifying multi-goal path planning for autonomous data collection. In: 2014 IEEE/RSJ International Conference on Intelligent Robots and Systems, IEEE, pp 2937–2942

Fakcharoenphol J, Harrelson C, Rao S (2007) The k -traveling repairmen problem. ACM Transactions on Algorithms 3(4):40–es

Feo TA, Resende MGC (1989) A probabilistic heuristic for a computationally difficult set covering problem. Operations Research Letters 8(2):67–71

Feo TA, Resende MGC, Smith SH (1994) A Greedy Randomized Adaptive Search Procedure for Maximum Independent Set. Operations Research 42(5):860–878

Fischetti M, Laporte G, Martello S (1993) The Delivery Man Problem and Cumulative Matroids. Operations Research 41(6):1055–1064

Frederickson GN, Wittman B (2012) Approximation Algorithms for the Traveling Repairman and Speeding Deliveryman Problems. Algorithmica 62(3-4):1198–1221, 0905.4444

Gentilini I, Margot F, Shimada K (2013) The travelling salesman problem with neighbourhoods: MINLP solution. Optimization Methods and Software 28(2):364–378

Godinho MT, Gouveia L, Pesneau P (2014) Natural and extended formulations for the Time-Dependent Traveling Salesman Problem. Discrete Applied Mathematics 164:138–153

Gouveia L, Voß S (1995) A classification of formulations for the (time-dependent) traveling salesman problem. European Journal of Operational Research 83(1):69–82

Gunawan A, Lau HC, Vansteenwegen P (2016) Orienteering Problem: A survey of recent variants, solution approaches and applications. European Journal of Operational Research 255(2):315–332

Hart JP, Shogan AW (1987) Semi-greedy heuristics: An empirical study. Operations Research Letters 6(3):107–114

Hoos HH, Stützle T (1998) Evaluating Las Vegas Algorithms - Pitfalls and Remedies. Proceedings of the Fourteenth Conference on Uncertainty in Artificial Intelligence pp 238–245, 1301.7383

Kulich M, Přeučil L, Miranda-Bront JJ (2014) Single robot search for a stationary object in an unknown environment. In: 2014 IEEE International Conference on Robotics and Automation (ICRA), IEEE, pp 5830–5835

Kulich M, Miranda-Bront JJ, Přeučil L (2017) A meta-heuristic based goal-selection strategy for mobile robot search in an unknown environment. Computers & Operations Research 84:178–187

Lucena A (1990) Time-dependent traveling salesman problem – the deliveryman case. Networks 20(6):753–763

Martin O, Otto SW, Felten EW (1991) Large-step Markov Chains for the Traveling Salesman Problem. Complex Systems 5:219–224

Méndez-Díaz I, Zabala P, Lucena A (2008) A new formulation for the Traveling Deliveryman Problem. Discrete Applied Mathematics 156(17):3223–3237

Mikula J (2021) Search for a static object in a known environment. Master's thesis, Czech Technical University in Prague, Faculty of Electrical Engineering

Miranda-Bront JJ, Méndez-Díaz I, Zabala P (2014) Facets and valid inequalities for the time-dependent travelling salesman problem. European Journal of Operational Research 236(3):891–902

Mjirda A, Todosijević R, Hanafi S, Hansen P, Mladenović N (2017) Sequential variable neighborhood descent variants: an empirical study on the traveling salesman problem. International Transactions in Operational Research

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

Mladenović N, Urošević D, Hanafi S (2013) Variable neighborhood search for the travelling deliveryman problem. 4OR 11(1):57–73

Naeni LM, Salehipour A (2019) A New Mathematical Model for the Traveling Repairman Problem. In: 2019 IEEE International Conference on Industrial Engineering and Engineering Management (IEEM), IEEE, pp 1384–1387

Reinelt G (1991) TSPLIB - A Traveling Salesman Problem Library. ORSA Journal on Computing 3(4):376–384

Resende MGC, Ribeiro CC (2016) Optimization by GRASP. Springer New York, New York, NY

Ribeiro CC, Rosseti I (2015) tttplots-compare: a perl program to compare time-to-target plots or general runtime distributions of randomized algorithms. Optimization Letters 9(3):601–614

Ribeiro CC, Rosseti I, Vallejos R (2009) On the use of run time distributions to evaluate and compare stochastic local search algorithms. In: Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)

Rios EFS (2016) Exploração de estratégias de busca local em ambientes cpu/gpu. PhD thesis, Universidade Federal Fluminense

Roberti R, Mingozzi A (2014) Dynamic ng-Path Relaxation for the Delivery Man Problem. Transportation Science 48(3):413–424

Sahni S, Gonzalez T (1976) P-Complete Approximation Problems. Journal of the ACM (JACM) 23(3):555–565

Salehipour A, Sörensen K, Goos P, Bräysy O (2011) Efficient GRASP+VND and GRASP+VNS metaheuristics for the traveling repairman problem. 4OR 9:189–209

Santana Í, Plastino A, Rosseti I (2020) Improving a state-of-the-art heuristic for the minimum latency problem with data mining. International Transactions in Operational Research 00:itor.12774, 1908.10705

Sarmiento A, Murrieta-Cid R, Hutchinson S (2004) A multi-robot strategy for rapidly searching a polygonal environment. In: Lecture Notes in Artificial Intelligence (Subseries of Lecture Notes in Computer Science), vol 3315, pp 484–493

Satyananda D, Wahyuningsih S (2019) Sequential order vs random order in operators of variable neighborhood descent method. Telkomnika (Telecommunication Computing Electronics and Control) 17(2):801–808

Silva MM, Subramanian A, Vidal T, Ochi LS (2012) A simple and effective metaheuristic for the Minimum Latency Problem. European Journal of Operational Research 221(3):513–520

Smith SL, Imeson F (2017) GLNS: An effective large neighborhood search heuristic for the Generalized Traveling Salesman Problem. Computers and Operations Research 87:1–19