Dragan Urošević

VARIABLE NEIGHBORHOOD SEARCH FOR DIFFERENT VARIANTS OF TRAVELING SALESMAN PROBLEM

Abstract. Variable neighborhood search (VNS) is a framework for building heuristics based upon systematic changes of neighborhoods both in a local search phase, to find a local minimum, and in a perturbation phase to escape from the large valley. In this chapter, we provide an overview of different VNS variants and describe how they can be used to solve some variants of the Traveling salesman problem. More precisely, we present different neighborhood structures that may be exploited, and show how they can be organized within variable neighborhood descent and variable neighborhood search heuristics. Finally, we provide insights on the performance of different VNS methodologies applied to two variants of the Traveling salesman problem.

Mathematics Subject Classification (2020): 05-02, 05C50; 99X99

Keywords: combinatorial optimization, variable neighborhood search, traveling salesman problem

Mathematical Institute of the Serbian Academy of Sciences and Arts, Belgrade, Serbia
draganu@mi.sanu.ac.rs
ORCID: 0000-0003-3607-6704

https://doi.org/10.64191/zr2510310105

Contents

1. Introduction 179
2. Variable neighborhood search methodology 179
2.1. Shaking procedure 180
2.2. Neighborhood change step 181
2.3. Improvement procedures used within VNS 182
2.4. Basic VNS 184
2.5. Reduced VNS 184
2.6. General VNS 185
2.7. Skewed VNS 186
3. Pick Up and Delivery Traveling Salesman Problem 186
3.1. Initial solution for 1-PDTSP 187
3.2. Neighborhoods for the 1-PDTSP 188
3.3. Local Search and Shaking 190
3.4. An efficient implementation of GVNS for 1-PDTSP 190
3.5. Computational results 195
3.6. Conclusions 197
4. Traveling Deliveryman Problem 198
4.1. Introduction 198
4.2. Problem formulation 198
4.3. k-opt complexity 201
4.4. General VNS for solving TDP 204
4.5. Initial solution 204
4.6. Shaking 205
4.7. VND algorithms for TDP 206
4.8. GVNS algorithms for TDP 207
4.9. Computational results for TDP 207
4.10. Conclusions 210
5. Conclusions 211
Acknowledgments 211
References 211

1. Introduction

An optimization problem may be stated in a general form as:

$$\min\{f(x)|x \in X \subseteq \mathcal{S}\} \tag{1.1}$$

where $\mathcal{S}, X, x$ and f respectively denote the solution space, the feasible set, a feasible solution and a real-valued objective function. Depending on the set $\mathcal{S}$, we may distinguish between combinatorial optimization problems where set $\mathcal{S}$ is discrete (finite, but usually extremely large, or infinite), and continuous optimization problems where $\mathcal{S} = \mathbb{R}^n$.

Let $\mathcal{N}(x)$ denote a neighborhood structure of a given solution x (i.e., $\mathcal{N} : X \to \mathcal{P}(X)$, where $\mathcal{P}(X)$ denotes the power set of the set X). Usually, a neighborhood $\mathcal{N}(x)$ is defined by using a given metric (or quasi-metric) function d introduced in the solution space $\mathcal{S}$ as:

$$\mathcal{N}(x) = \{y \in X|d(x, y) \leqslant \alpha\}$$

where α is a given positive number. Then, on the one hand, a solution $x^* \in X$ is a local minimum, relative to neighborhood $\mathcal{N}(x^*)$, for problem (1.1) if $f(x^*) \leqslant f(x),\ \forall x \in \mathcal{N}(x^*)$. On the other hand, a solution $x^* \in X$ is an optimal solution (global optimum) for problem (1.1) if $f(x^*) \leqslant f(x),\ \forall x \in X$.

Many practical instances of problems of form (1.1) cannot be solved exactly (i.e., by providing an optimal solution with the proof of its optimality) in reasonable time. It is well-known that many problems of the form (1.1) are NP-hard, i.e., no algorithm with a number of steps that is polynomial in the size of the instances is known for solving any of them, and that if one were found, it would be a solution for all. Because of that, there is a need for so called heuristics which are able to relatively quickly produce an approximate solution of high quality (i.e solution very close to optimal), or sometimes an optimal solution but without proof of its optimality.

Note that there could be a very large number of local minima for some optimization problems. Usually, it is not so hard to get a local minimum, but it is not easy to find a global one. Indeed, such search methods may get trapped in a local optimum and miss the global one. Resolving local optima traps is an important issue, which requires finding a way to escape from any local optimum found by the solution method.

The rest of this chapter is organized as follows. In Section 2, we provide the description of VNS in general terms. After that, Section 3 presents VNS methodology for solving Pick Up and Delivery Traveling Salesman Problem. In Section 4 VNS methodology for solving Traveling Deliveryman Problem is presented. Finally, some conclusions regarding possibility of using VNS metaheuristic for solving different variant of Traveling Salesman Problem are presented in Section 5.

2. Variable neighborhood search methodology

Variable neighborhood search (VNS) is a metaheuristic proposed by Mladenović and Hansen [30]. It presents a flexible framework for building heuristics for approximately solving combinatorial and continuous optimization problems. VNS

systematically changes neighborhood structures during the search for an optimal (or near-optimal) solution based on the following observations:

(1) A local optimum relative to one neighborhood structure is not necessarily a local optimum for another neighborhood structure;
(2) A global optimum is a local optimum with respect to all neighborhood structures;
(3) Empirical evidence shows that for many problems, all or a large number of the local optima are relatively close to each other.

The first property is exploited by increasingly using complex moves (which define different neighborhood structures) in order to find local optima with respect to all neighborhood structures used. The second property suggests using several neighborhoods, if local optima found are of poor quality. Finally, the third property suggests increased intensity of the search in the region of solution space around the current incumbent solution.

The work of a VNS heuristic, in a very general form, consists in applying the three following subroutines:

(1): Shaking (or perturbation) procedure
(2): Improvement procedure
(3): Neighborhood change step

cyclically until a predefined stopping criterion is met. The aim of the shaking procedure and neighborhood change step is to guide the search toward new unexplored parts of the feasible region in order to diversify the search, while the improvement procedure is an intensified search from a perturbed solution that hopefully leads to a new best found solution. In other words, the improvement procedure is used to generate local optima, while the shaking and neighborhood change procedures are used to resolve local optima traps.

VNS based heuristics have been successfully applied to a wide range of optimization problems, for example: diversity/dispersion problems [2,5,33], clustering problems [8,9], location [6,7,28,34], routing [32,36], scheduling problems [23,24,37,38].

The most recent surveys on different VNS and VND variants may be found in [10,15,29].

In the rest of this section, we provide more details about the shaking procedure, neighborhood change step and different improvement procedures that may be used within the VNS framework. After that we provide an overview of VNS variants derived by combining the shaking procedure, neighborhood change step and improvement procedures in different ways.

2.1. Shaking procedure. As already said, the aim of the shaking procedure used within a VNS heuristic is to hopefully resolve local optima traps. The standard shaking procedure consists of selecting a random solution from the k^{th} neighborhood of a current (incumbent) solution. For some problem instances, a completely random jump in the k^{th} neighborhood is too diversified. Hence, sometimes it is preferable to do a so-called intensified shaking that takes into account how sensitive the objective function is to small changes (shaking) of the solution. However, for

the sake of simplicity, we shall present here only a simple shaking procedure based on selecting a random solution from the k^{th} neighborhood (see Algorithm 1).

Algorithm 1: Shaking procedure.

Function SHAKE$(x, k, \mathcal{N})$;
choose $x' \in \mathcal{N}_k(x)$ at random;
return x'

2.2. Neighborhood change step. The purpose of a neighborhood change step is to guide the variable neighborhood search heuristic while it explores the solution space. More precisely, it makes a decision in which neighborhood to explored next, as well as whether some solution will be accepted as a new incumbent solution or not. The most widely used neighborhood change procedures may be found in [15]:

- SEQUENTIAL NEIGHBORHOOD CHANGE STEP-If an improvement of the incumbent solution does not occur in some neighborhood, the search is resumed in the next neighborhood (according to a defined order) of the incumbent solution; otherwise, a move is made to a new incumbent solution and the neighborhood number (k) is reset to 1 (see Algorithm 2).

Algorithm 2: Sequential neighborhood change step.

Procedure NEIGHBORHOOD_CHANGE_SEQUENTIAL(x, x', k)
$f(x') < f(x) \quad x \leftarrow x'$;
$k \leftarrow 1$;
$k \leftarrow k + 1$;

- The SKEWED NEIGHBORHOOD CHANGE STEP accepts as new current solutions not only improving ones, but also worse solutions than the current incumbent if a certain criterion is met. The purpose is to allow exploration of valleys far from the incumbent solution. Because of that, in the *skewed neighborhood change step*, a trial solution (y) is evaluated taking into account not only the objective values of the trial and the incumbent solution (x), but also the distance between them. The evaluation function used in the skewed neighborhood change step may be stated as:

$$g(x, y, \alpha) = \frac{f(y)}{f(x)} - \alpha \cdot \frac{d(x, y)}{d_{\max}},$$

where $d(x, y)$ is the distance between solutions x and y (e.g., Hamming distance) and $d_{\max}$ is the maximal possible distance between any two solutions (or $k_{\max}$ if distances in the solution space can be arbitrarily large, i.e., for any $D \geqslant 0$, there exist solutions x and y such that $d(x, y) > D$), and α is parameter depending on solved problem. The *skewed neighborhood change step* using this function and sequential neighborhood change is given in Algorithm 3.

Algorithm 3: Neighborhood change step for skewed VNS.

Procedure SKEWED_NEIGHBORHOOD_CHANGE(x, x', k, α)
if $g(x, x', \alpha) < 1$ **then**
  $x \leftarrow x'$;
  $k \leftarrow 1$;
else
  $k \leftarrow k + 1$;
end

2.3. Improvement procedures used within VNS.

In this section we present improvement procedures commonly used within a VNS heuristic. They range from a simple local search within a single neighborhood to more advanced variable neighborhood descent procedures that explore multiple neighborhoods.

2.3.1. Local search. A local search heuristic is based on the exploration of a single neighborhood structure $N(x)$ applied to current solution x at each iteration. Starting from an initial solution x, it selects at each iteration a better solution x' (if any) from the predefined neighborhood structure $N(x)$, and sets it to be the new incumbent solution ($x \leftarrow x'$). A local search heuristic finishes its work when reaching an incumbent solution x that is already a local optimum with respect to its neighborhood structure $N(x)$. The most common search strategies used within a local search heuristic are the *first improvement* search strategy (as soon as an improving solution in a neighborhood structure $N(x)$ is detected, it is set to be the new incumbent solution), and the *best improvement* search strategy (the best among all improving solutions in $N(x)$ (if any) is set to be the new incumbent solution). The steps of a local search heuristic using the first improvement strategy and a local search heuristic using the best improvement strategy are given in Algorithms 4 and 5, respectively.

2.3.2. Variable neighborhood descent procedures. The Variable Neighborhood Descent (VND) procedure exploits the fact that a solution which is a local optimum with respect to several neighborhood structures is more likely to be a global optimum than a solution generated as a local optimum for just one neighborhood structure. More precisely, a VND procedure explores several neighborhood structures either in a sequential or nested fashion in order to (possibly) improve a given solution. As a search strategy, it may use either first improvement or best improvement in any given neighborhood.

Sequential VND. (Seq-VND) procedure works in the following way. Several neighborhood structures are ordered first in a list, and after that, examined one after another respecting the established order. Let $N = \{N_1, \ldots, N_{\ell_{\max}}\}$ be a set of operators such that each operator N_ℓ, $1 \leqslant \ell \leqslant \ell_{\max}$ maps a given solution x to a predefined neighborhood structure $N_\ell(x)$. Note that the order of operators in the set N will also define the order of examining neighborhood structures of a given solution x. Starting from a given solution x, the sequential VND procedure

Algorithm 4: Local search using the first improvement search strategy.

```
Function LS_FI(x, N)
repeat
    Let 𝒩(x) = {x^1, ..., x^p};
    i ← 0;
    x' ← x;
    repeat
        i ← i + 1;
        if f(x^i) < f(x) then
            x ← x^i
            break;
        end
    until i = p;
until f(x') ⩾ f(x);
return x;
```

Algorithm 5: Local search using the best improvement search strategy.

```
Function LS_BI(x, N)
repeat
    x' ← x;
    x ← argmin_{y∈N(x')} f(y);
until f(x') ⩽ f(x);
return x';
```

iteratively explores its neighborhood structures defined by the set N, one after another according to the established order. As soon as an improvement of the incumbent solution in some neighborhood structure occurs, the sequential VND procedure resumes the search in the first neighborhood structure (according to the defined order) of the new incumbent solution. The whole process is stopped if the current incumbent solution cannot be improved with respect to any of the $\ell_{\max}$ neighborhood structures. The steps of sequential VND using the best improvement search strategy are given in Algorithm 6.

Note that each of these VND algorithms may also use the first improvement search strategy to explore neighborhood structures. If the first improvement strategy is used, the steps of each VND variant are the same as the steps of the corresponding VND variant that uses the best improvement strategy, but with the additional assumption that call $\operatorname{argmin}_{y \in N_\ell(x)} f(y)$ returns the first encountered solution y in the neighborhood $N_\ell(x)$ such that $f(y) < f(x)$.

Nested VND. Another possibility is to explore the neighborhoods with a nested strategy (Nest-VND). The pure nested strategy may be described generally as follows. Assume that we define two neighborhood structures ($\ell_{\max} = 2$). In nested VND we in fact perform local search with respect to the first neighborhood in any

Algorithm 6: Sequential VND using the best improvement search strategy.

Function SEQ-VND(x, $\ell_{\max}$, N)
repeat
  $stop \leftarrow$ `false`;
  $\ell \leftarrow 1$;
  $x' \leftarrow x$;
  repeat
    $x'' \leftarrow \operatorname{argmin}_{y \in N_\ell(x)} f(y)$;
    NEIGHBOORHOOD_CHANGE_SEQUENTIAL(x, x'', ℓ);
  until $\ell > \ell_{\max}$;
  if $f(x') \leqslant f(x)$ **then**
    $stop \leftarrow$ `true`;
  end
until $stop$=`true`;
return x';

point of the second. If both neighborhoods have cardinality n, then with Seq-VND we visit $2n$ points, but in Nest-VND we visit n^2 points. Therefore, pure Nest-VND neighborhoods are much larger. The nested VND was applied for the first time to solve the uncapacitated single allocation p-hub median problem (see [19])

It is important to note that a local minimum in Seq-VND will not necessarily be a local minimum in Nest-VND, but the opposite holds if the Nest-VND is designed to contain the Seq-VND neighborhood. Thus, the number of local minima in Nest-VND will in general be smaller than in Seq-VND. Of course, the larger neighborhood will require more computational time. See Algorithm 7 for a detailed pseudo-code of Nest-VND.

Mixed VND. There are several ways to reduce this large nested set of neighborhood points. One way is to reduce the size of the largest neighborhoods, or, after b (a parameter) neighborhoods, switch from a nested to a sequential strategy. The reduced Nested VND will be called Mix-VND.

2.4. Basic VNS.

The basic VNS variant executes alternately shaking procedure presented in Algorithm 1, and a simple local search procedure (one of the two presented in Algorithms 4 and 5), together with a neighborhood change step (this one presented in Algorithm 2) until a predefined stopping criterion is met. The most commonly used stopping criterion is a specified limit on CPU time. The steps of Basic VNS using the sequential neighborhood change step are given in Algorithm 8.

2.5. Reduced VNS.

From this basic VNS scheme, several other VNS approaches may be derived. The simplest one is called *Reduced VNS*, which combines a shaking procedure and a neighborhood change step while discarding the improvement phase. The steps of *Reduced VNS* with sequential neighborhood change function are given in Algorithm 9.

Algorithm 7: Nest-VND: Nested Variable neighborhood descent heuristic.

Function NEST-VND(x, $\ell_{\max}$, N)
$x_{opt} \leftarrow x$, $f_{opt} \leftarrow f(x)$;
$\ell \leftarrow \ell_{\max}$;
repeat
  if all solutions from *neighborhood* N_ℓ *are visited* **then**
    $\ell \leftarrow \ell + 1$;
  end
  if *there is any non visited solution* $x_\ell \in N_\ell(x)$ **then**
    $x_{cur} \leftarrow x_\ell$;
  end
  if $\ell \geqslant 2$ **then**
    $\ell \leftarrow \ell - 1$;
  else
    $f_{cur} \leftarrow f(x_{cur})$;
    if $f_{cur} < f_{opt}$ **then**
      $x_{opt} \leftarrow x_{cur}$; $f_{opt} \leftarrow f_{cur}$;
    end
  end
until $\ell > \ell_{\max}$ *(i.e., there are no more points in the last neighborhood)*;
return x_{opt};

Algorithm 8: Basic Variable Neighborhood Search.

Function BASIC_VNS(x, $k_{\max}$, $\mathcal{N}$,N)
repeat
  $k \leftarrow 1$;
  while $k \leqslant k_{\max}$ *and stopping condition is not fulfilled* **do**
    $x' \leftarrow$ SHAKE$(x, k, \mathcal{N})$;
    $x'' \leftarrow$ LOCAL_SEARCH(x', N);
    NEIGHBORHOOD_CHANGE_SEQUENTIAL(x, x'', k);
  end
until *stopping condition is fulfilled;*
return x;

2.6. General VNS. Another variant known as *General VNS* uses some VND procedure presented above, unlike the improvement phase in Basic VNS, which uses just a simple local search. Note that it is not obligatory that neighborhood structures used within the shaking procedure and the VND procedure be the same although it can be a desirable feature. The steps of General VNS are given in Algorithm 10. Here the statement VND(x', $\ell_{\max}$, N) means that a certain VND variant presented above is executed.

Algorithm 9: Reduced Variable Neighborhood Search.

Function REDUCED_VNS(x, $k_{\max}$, $\mathcal{N}$)
repeat
  $k \leftarrow 1$;
  while $k \leqslant k_{\max}$ *and stopping condition is not fulfilled* **do**
    $x' \leftarrow$ SHAKE$(x, k, \mathcal{N})$;
    NEIGHBORHOOD_CHANGE_SEQUENTIAL(x, x', k);
  end
until *stopping condition is fulfilled;*
return x;

Algorithm 10: General Variable Neighborhood Search.

Function GENERAL_VNS(x, $k_{\max}$, $\ell_{\max}$, $\mathcal{N}$,N)
repeat
  $k \leftarrow 1$;
  while $k \leqslant k_{\max}$ *and stopping condition is not fulfilled* **do**
    $x' \leftarrow$ SHAKE$(x, k, \mathcal{N})$;
    $x'' \leftarrow$ VND(x', $\ell_{\max}$, N);
    NEIGHBORHOOD_CHANGE_SEQUENTIAL(x, x'', k);
  end
until *stopping condition is fulfilled;*
return x;

2.7. Skewed VNS. Skewed VNS variants are those that in the neighborhood change step accept as new incumbent solutions not only improving solutions but in some cases those which are worse than the current incumbent solution. More precisely, General Skewed Variable Neighborhood Search also accepts non-improving solutions that are "slightly" worse than the incumbent one, but at the same time "significantly" far from the incumbent solution in the solution space. In this way General Skewed VNS continues the search in parts of the solution space far away from the incumbent solution while starting from a solution which is only slightly worse according to the value of the objective function. Therefore, in the *skewed neighborhood change step*, a trial solution is evaluated taking into account not only the objective values of the trial and the incumbent solution but also the distance between them as shown in Algorithm 3. In that way, Skewed VNS may successfully resolve local optima traps by exploring promising regions far from the incumbent solution.

3. Pick Up and Delivery Traveling Salesman Problem

The one-commodity pickup-and-delivery travelling salesman problem (1-PDTSP for short) was introduced by Hernández-Pérez [16].

Problem description. 1-PDTSP is an extension of the classical TSP. It involves a set of locations with specified travel distances. One location acts as a depot, while

others are customers categorized into delivery (consumers) and pickup (producers). A single commodity needs transport between customers, each having specific demand or supply requirements. A vehicle with fixed capacity starts and ends at the depot, visiting each customer exactly once. The vehicle must not exceed capacity after pickups and can serve delivery customers if loaded sufficiently. The goal is to find a minimum Hamiltonian route meeting all customer needs, with flexibility on the vehicle's initial load.

The 1-PDTSP has various applications in routing a single commodity through a circular network represented as a graph connecting multiple sources and destinations. A practical example is demonstrated by Anily and Bramel [3] in the context of inventory repositioning. Consider a set of retailers owned by the same company and located at different sites. Due to the random nature of demand, some retailers may experience inventory surpluses while others face shortages. If the company decides to redistribute inventory from surplus retailers to those in need, the problem of determining the most cost-effective Hamiltonian route for a capacitated vehicle aligns with the 1-PDTSP. This represents a simplified version of more complex scenarios involving multiple commodities or multiple capacitated vehicles.

Notation Let us introduce the notation used throughout rest of this part. Let $G = (V, E)$ be a complete and undirected graph with vertex set $V = \{1, 2, \ldots, n\}$. Vertex 1 represents the depot. The other vertices from 2 to n represent customers. Each vertex is associated with non–zero demand q_i. If $q_i > 0$, then customer i is a pickup customer (or producer which produce q_i units of product), otherwise, it is a delivery customer (or consumer that requires $-q_i$ units of product). The depot can be considered as a customer whose demand $q_1 = -\sum_{i=2}^{n} q_i$. If $q_1 > 0$ then q_1 represents the initial load of the vehicle, otherwise $-q_1$ represents the final load after visiting all customers. In other words, the pickup/delivery demand at the depot balances the other pickup/delivery demands.

The travel cost $c(i, j)$ for each edge $(i, j) \in E$ is given. It can be assumed that travel costs between locations are proportional to their distances. The capacity of the vehicle is denoted by Q ($Q > 0$). A vehicle with the capacity Q must visit each customer only once, and its capacity must not be exceeded after visiting a pickup customer and its load must be bigger than (or equal to) the demand of a delivery customer before the visit. The 1-PDTSP should determine the tour that starts and ends at the depot and has a minimum length. Note that typically it holds

$$\max_{i \in V} |q_i| \leqslant Q \leqslant \max \left\{ \sum_{i \in V,\ q_i > 0} q_i, - \sum_{i \in V,\ q_i < 0} q_i \right\}.$$

The left inequality means that no feasible tour can be found if the vehicle capacity is less than the maximum customer demand. The second inequality makes this problem different from the standard TSP.

3.1. Initial solution for 1-PDTSP.

The solution space of the 1-PDTSP consists of all possible tours starting at the depot. Each tour is represented as a permutation of the set $1, \ldots, n$, where the first element, labeled 1, represents the depot.

However, many tours may be infeasible if the total commodity collected from consecutive pickup customers exceeds the vehicle's capacity or if the total required by consecutive delivery customers surpasses the available load. The process begins with a potentially infeasible tour, and once a feasible tour is identified, the search is confined to the space of feasible tours.

Let $x = (x_1, x_2, \ldots, x_n)$ denote a tour, each customer being visited exactly once, with $x_1 = 1$ and let $L_i(x)$ be the load of the vehicle after visiting the customer x_i, i.e. i-th customer in the tour x,

$$L_1(x) = q_{x_1} = q_1 \quad \text{and} \quad L_i(x) = L_{i-1}(x) + q_{x_i}.$$

Then, as shown in [17], x is a feasible tour of 1-PDTSP if and only if

$$\max_{i \in V} L_i(x) - \min_{i \in V} L_i(x) \leqslant Q. \tag{3.1}$$

The method presented in [40] is used to generate the initial solution because it generally produces better results than the algorithm described in [18]. This constructive procedure begins by randomly selecting the first customer, x_2. The next customer, x_{i+1}, is chosen from a data structure containing the $ncl = 20$ closest unvisited customers (i.e., customers not yet included in the tour x) to the last inserted customer, x_i. Only feasible sub-tours are considered, ensuring that the capacity constraints for both delivery and collection are not violated after inserting x_{i+1}. Among these ncl closest customers, we extract those that can be added feasibly to the end of the tour x and select the customer with the highest demand. This choice is based on the belief of the authors [40] that selecting customers with higher demand increases the chance for obtaining a feasible tour. If no such customer exists, we look at all unvisited customers and determine S as the set of customers that can be feasibly added to the sub-tour. From S, we choose the nearest customer with a probability of 90%, or randomly select one from S with a probability of 10%. If no customers can be feasibly inserted ($S = \emptyset$), we insert a random customer and proceed. It's important to note that this algorithm may result in an infeasible solution. In such cases, it is recommended repeating the process until a feasible solution is found [40]. But, in our GVNS, we accept the infeasible solution as the initial one.

3.2. Neighborhoods for the 1-PDTSP. In a local search algorithm, a collection of neighborhoods is created by defining moves that allow transition from one (feasible) solution to another. The basic neighborhoods for the classical TSP, which involve edge exchanges and node insertion moves, can be effectively extended to the 1-PDTSP. A well-known heuristic of this type is the k-exchange [25], also known as k-opt, which has been successfully applied in local searches for various problems such as assignment, routing, and scheduling.

The 2–opt procedure is a straightforward method. It removes two non–consecutive edges from the tour and reconnects the resulting two paths. There is only one valid way to reconnect these paths, ensuring the tour remains valid (for classical TSP). This move is applied only if the resulting tour is shorter, and the process of removing and reconnecting edges continues until no further 2–opt improvements can be

made. In the neighbor–list implementation of 2–opt, we first sort all customers in non–decreasing order based on their distances as a preprocessing step, effectively creating a ranked matrix. For each edge $e_i = (x_i, x_{i+1})$, we identify the edges $e_j = (x_j, x_{j+1})$ that satisfy the condition:

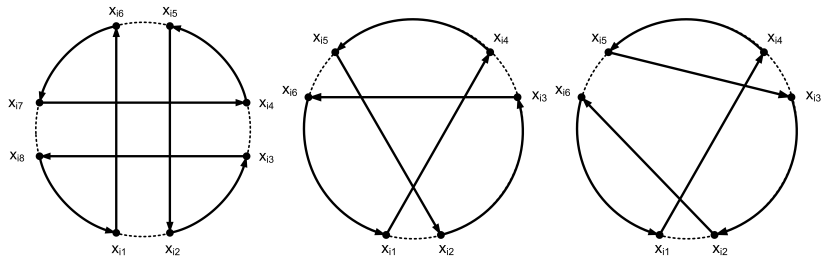
$$c(x_i, x_{i+1}) + c(x_j, x_{j+1}) > c(x_i, x_j) + c(x_{i+1}, x_{j+1}).$$

For each fixed edge e_i, customers x_j are considered in non-decreasing order of their distance from x_i, and similarly, customers x_{j+1} are considered based on their distance from x_{i+1}. The processing can be halted if $c(x_i, x_j) > c(x_i, x_{i+1})$, or similarly, processing of vertices based on their distance from x_{i+1} can be stopped if $c(x_{i+1}, x_{j+1}) > c(x_{i+1}, x_i)$.

The k–opt neighborhood is an extension of the 2–opt method. It involves removing k non–consecutive edges and inserting k new edges to form a new tour. There are multiple ways to reconstruct a feasible tour after removing k edges. Additionally, the number of neighboring solutions in a k–opt neighborhood grows as $O(n^k)$ [26], making the process of exploring all possible k–opt neighboring solutions computationally expensive. As a result, restricted k–opt neighborhoods are commonly used. In our method, in addition to k–opt, we decide to use Insertion and Double bridge moves (neighborhoods):

(i): Insertion is a specific case of 3–opt, where each customer is inserted between any two other customers. Its complexity is $O(n^2)$;

(ii): Double bridge [22] is a specific case of 4–opt. This method is crucial for some combinatorial problems, as it preserves the orientation of the tour.



(a) (b) (c)

FIGURE 1. Double–bridge (a) and 3–opt without change of direction (b) and with change of direction (c).

In addition to 2–opt, the GVNS framework also uses the Insertion, 3–opt, and double–bridge neighborhoods. Furthermore, the Insertion neighborhood is divided into two distinct subsets: forward insertion and backward insertion.

The *forward insertion* move aims to find a shorter tour by relocating a customer from its current position i to a new position j, where $j > i$. This requires shifting all customers on positions $i+1, i+2, \ldots, j$ one position backward, while the customers on positions 1 to $i-1$ and $j+1$ to n remain unchanged.

The *backward insertion* move operates similarly, but here the selected customer on position i is moved to a position j where $j < i$. In this case, the intermediate customers are shifted one position forward.

3.3. Local Search and Shaking. A complete local search is organized as a Variable Neighborhood Descent (VND) through the 2–opt neighborhood (N_1), forward and backward insertion neighborhoods (N_2 and N_3), meaning the parameter $\ell_{\max}$ in Algorithm 6 is set to 3. Additionally, we developed another local search method called Mix-VND. Let x represent the current solution. In this approach, we randomly select a solution x' from the 3–opt or double–bridge neighborhoods of x and then apply Seq-VND (which includes 2–opt, forward, and backward insertion neighborhoods) starting from x'. If the resulting solution x'' improves upon the current solution x, we adopt it and re-center the search around x'', updating $x \leftarrow x''$. If not, we repeat the process by randomly choosing a new solution and applying Seq-VND again. The maximum number of unsuccessful trials is set to 200. As a result, this local search is not fully deterministic.

Shaking is performed using two well–known neighborhoods commonly used for solving the Traveling Salesman Problem (TSP): 3–opt and double–bridge. A random point in the $\mathcal{N}_k$ neighborhood is obtained by executing k_{3-opt} 3–opt moves and k_{db} double–bridge moves. Here, k_{3-opt} is randomly selected from the range $[k/2, k]$, and $k_{db} = k - k_{3-opt}$, where $k = 1, 2, \ldots, k_{\max}$. These moves are performed while ensuring feasibility, provided the current solution is feasible. If maintaining feasibility is not possible, a random point is generated from either the 3–opt or double–bridge neighborhood.

3.4. An efficient implementation of GVNS for 1-PDTSP. The key implementation difference between the 1–PDTSP and the TSP is that some customer permutations in the 1–PDTSP do not represent feasible tours. As a result, an efficient data structure is required to check the feasibility of proposed tours. This section explains how feasibility checks are implemented for all neighborhoods used in the VND, specifically within the GVNS framework.

3.4.1. 2-opt with feasibility checking. As discussed earlier, we use the neighbor-list implementation of 2-opt [12,26], also referred to as the sequential implementation in [21]. Efficiently exploring the 2-opt neighborhood requires evaluating the feasibility of each solution within the neighborhood. This requires a data structure that allows efficient feasibility checking and straightforward updates. A suitable data structure was introduced in [20], with worst-case complexity of $\mathcal{O}(n^{4/3})$ for updates, $\mathcal{O}(1)$ for feasibility checks, and a memory requirement of $\mathcal{O}(n^{4/3})$.

To enhance efficiency, we propose to use a *Binary Segment Tree* data structure. This approach enables feasibility checks for neighbor tours in $\mathcal{O}(\log n)$ time. If a better solution is identified, the segment tree structure can be updated or reconstructed in $\mathcal{O}(n)$ time.

The Binary Segment Tree, introduced by [11], is an efficient data structure for maintaining cumulative frequencies or partial sums. It leverages the fact that any integer interval can be expressed as a union of intervals with lengths that are powers

of two. This allows cumulative frequencies over an interval to be represented as the sum of corresponding subsets of "sub-frequencies". Operations are performed using the binary representation of indices.

Let L be an array of n elements. The Binary Segment Tree supports the following basic operations:

- for a given value v and index i, add v to the element L_i, $1 \leqslant i \leqslant n$;
- for a given interval $[i, j]$, find maximum/minimum among values $L_i, L_{i+1}, \ldots, L_j$, $1 \leqslant i \leqslant j \leqslant n$.

The data structure is stored in an array called $Tree$ (see Algorithm 11 and Algorithm 12 for the case of computing the maximum). This structure is space-efficient, requiring only as much storage as a simple array with n elements. For vehicle loads, we use a Binary Segment Tree to efficiently compute the maximum and minimum values within intervals of these loads.

The $Tree$ structure forms a complete binary tree with the root node at position 1. The leaves correspond to the elements of the array L, arranged from left to right at the tree's last level. These elements are stored in positions ranging from 2^p to $2^p + n - 1$, where p is the smallest integer satisfying $2^p \geqslant n$, representing the tree's height.

The internal nodes of $Tree$ store aggregated values (e.g., sum, max, or min) of the leaf nodes within their respective subtrees. In this case, we utilize only min and max. Specifically, the value of an internal node i is the cumulative value of its two children. The parent of node i is located at $\lfloor i/2 \rfloor$, while its left and right children are at $2 \cdot i$ and $2 \cdot i + 1$, respectively.

By design, the Segment Tree contains at most $2 \cdot n$ nodes, and its height is $\lceil \log_2(2 \cdot n) \rceil$.

To update, we traverse the vertices from the leaf node storing value of update element to the root, updating the values of the parent vertices along the way. For querying, we examine the parents of the left (i) and right (j) vertices and their respective intervals, then iteratively compute the max/min over the range $[i, j]$. Notably, the max/min operations can be replaced with any associative operation, such as sum, product, greatest common divisor (gcd), lowest common multiplier (lcm), etc. The Binary Segment Tree is updated by setting the element L_i to the new value v, as described in Algorithm 11.

Our algorithms make use of efficient binary operators. The bitwise exclusive OR operator (**xor**), utilized in Algorithm 12, compares each bit of the first operand with the corresponding bit of the second operand. If one bit is 0 and the other is 1, the resulting bit is set to 1; otherwise, it is set to 0. Similarly, the bitwise AND operator (**and**), also employed in these algorithms, compares each bit of the two operands. If both bits are 1, the resulting bit is set to 1; otherwise, it is set to 0. Additionally, Algorithm 11 makes use of the **div** operator for performing integer division of the left-hand operand by the right-hand operand.

In [31] the following properties of the Binary Segment Tree structure are proven:

- Updating the Binary Segment Tree after setting the element L_i to the new value is executed in $\mathcal{O}(\log n)$.

Algorithm 11: Updating the Binary Segment Tree, after setting the element L_i to the new value v.

Input: The value v, element index i and $p = \max\{s : 2^s < n\}$.
$i \leftarrow i + 2^{p+1} - 1$;
$Tree[i] \leftarrow v$;
while $i > 0$ **do**
    $Tree[i \textbf{ div } 2] \leftarrow \max(Tree[i], Tree[i \textbf{ xor } 1])$;
    $i \leftarrow i \textbf{ div } 2$;

Algorithm 12: Calculating the maximum value in interval $L_i, L_{i+1}, \ldots, L_j$.

Input: The parameters i and j are bounds of interval $[i, j]$ and $p = max\{s : 2^s < n\}$.
Output: The maximum value among elements $L_i, L_{i+1}, \ldots, L_j$.
$i \leftarrow i + 2^{p+1} - 2$; $j \leftarrow j + 2^{p+1}$;
$v_{\max} \leftarrow -1$;
while $(i \textbf{ div } 2) \neq (j \textbf{ div } 2)$ **do**
    if $(i \textbf{ and } 1) = 0$ **then** $v_{\max} \leftarrow \max(v_{\max}, Tree[i+1])$;
    $i \leftarrow i \textbf{ div } 2$;
    if $(j \textbf{ and } 1) \neq 0$ **then** $v_{\max} \leftarrow \max(v_{\max}, Tree[j-1])$;
    $j \leftarrow j \textbf{ div } 2$;
return $v_{\max}$.

- Updating the complete Binary Segment Tree, after setting the elements of array L to new values, is executed in $\mathcal{O}(n)$.
- Calculating the maximum (or minimum) value in interval $[L_i, L_j], j \geqslant i$ is executed in $\mathcal{O}(\log n)$ time.

Based on these properties, the following very significant property can be proven.

Property 3.1. *Checking the feasibility of the* 2*–opt move for 1–PDTSP with the Binary Segment Tree structure is in* $\mathcal{O}(\log n)$.

Proof. Let $x = (x_1, \ldots, x_n)$ be a current tour and $L = (L_1, \ldots, L_n)$ its associated load vector (where $L_i = L_i(x)$ is the vehicle load after visiting the i-th customer in the tour x). We will use Equation (3.1) for checking the feasibility of the solution. Let $Tree_{\max}$ and $Tree_{\min}$ be the Binary Segment Tree structures for computing the maximum and minimum vehicle loads for each city. After performing the 2–opt move with attributes i and j associated with the edges $e_i = (x_i, x_{i+1})$ and $e_j = (x_j, x_{j+1})$ with $1 \leqslant i < j \leqslant n$, the new tour x' looks

$$x_1 \to x_2 \to \ldots \to x_i \to x_j \to x_{j-1} \to \ldots \\ \to x_{i+1} \to x_{j+1} \to x_{j+2} \to \ldots \to x_{n-1} \to x_n \to x_1.$$

or

$$x'_k = \begin{cases} x_k & \text{for } k = 1, \ldots, i, \\ x_{i+j-k+1} & \text{for } k = i+1, \ldots, j \\ x_k & \text{for } k = j+1, \ldots, n. \end{cases}$$

Thus, only the vehicle loads for the customers $x_{i+1}, x_{i+2}, \ldots, x_j$ are changed. More precisely, the new load L' associated with the new solution x' is computed as follows:

$$L'_k = \begin{cases} L_k & \text{for } k = 1, \ldots, i, \\ L_i + \sum_{h=i+1}^{k} q_{x_{i+j-h+1}} & \text{for } k = i+1, \ldots, j \\ L_k & \text{for } k = j+1, \ldots, n. \end{cases}$$

Since $L_0 = 0$ and $\sum_{i=1}^{n} q_{x_i} = 0$, it is easy to conclude that we have

$$L_i + \sum_{h=i+1}^{k} q_{x_{i+j-h+1}} = L_i + L_j - L_{k-1} \qquad \text{for} \qquad k = i+1, i+2, \ldots, j-1, j.$$

To determine the new minimum vehicle load, we find the maximum load among $L_{j-1}, L_{j-2}, \ldots, L_i$ and subtract it $L_i + L_j$. This can be computed in $\mathcal{O}(\log n)$ time, using the $Tree_{\min}$ structure for the interval $[i, j-1]$. Analogously, we calculate the new maximum vehicle load and finally check the feasibility of the new tour x'. ■

3.4.2. Insertion feasibility checking. For the feasibility checking of Insertion (forward and backward) moves, the Binary Segment Tree structure is also used.

Property 3.2. *Feasibility checking of insertion moves for 1–PDTSP is in* $\mathcal{O}(\log n)$ *time.*

Proof. Let us divide insertion moves into forward and backward insertion moves. We prove here the property for the forward insertion move. We traverse the customers i from 1 to n in this order. After moving the customer i to the position $j > i$, we have the following loads

$$L_1, L_2, \ldots, L_{i-1}, L_{i+1} - q_{x_i}, L_{i+2} - q_{x_i}, \ldots, L_{j-1} - q_{x_i}, L_j - q_{x_i}, L_j, L_{j+1}, \ldots, L_n.$$

Of course, we need to calculate the maximum and minimum values ofthe new loads. It can be done by calculating $\max\{L_1, \ldots, L_{i-1}\}$, $\max\{L_{i+1}, \ldots, L_j\} - q_{x_i}$ and $\max\{L_j, \ldots, L_n\}$ as well as by calculating the minimum. If we use Binary Segment Tree structure, it can be done in $\mathcal{O}(\log n)$. ■

The overall complexity for an Insertion local search is $\mathcal{O}(n^2)$.

3.4.3. Maintaining the feasibility of 3-opt and Double-bridge moves. Once feasibility is achieved, it is important to maintain it during both the local search and Shaking steps of GVNS. The following two properties provide sufficient conditions to ensure the feasibility of random moves within the 3–opt and double–bridge neighborhoods.

The size of the 3–opt neighborhood is $\mathcal{O}(n^3)$. Once three arbitrary edges from the current tour are removed, there are 8 possible ways to insert three new edges to obtain a new tour. One of them, which does not change the orientation, is presented

in Figure 1. We denote this special 3–opt by 3–opt*. Assume that we choose three non-consecutive customers from the tour $x_{i_1}, x_{i_3}, x_{i_5}$, and let $x_{i_2}, x_{i_4}, x_{i_6}$ be their successors on the tour, respectively. Without loss of generality, assume that the depot is located between the customers x_{i_6} and x_{i_1}.

Property 3.3. *If $L_{i_1} = L_{i_3} = L_{i_5}$ holds for the vehicle loads, then we can perform the 3–opt* modification without violating the feasibility condition.*

Proof. It is enough to show that the loads for the customers involved in the move remain the same. For example, the new loads for the vertices x_{i_4}, x_{i_2}, x_{i_6} remain the same,

$$\begin{aligned}
L'_{i_4} &= q_{x_1} + \ldots + q_{x_{i_1}} + q_{x_{i_4}} = L_{i_1} + q_{x_{i_4}} = L_{i_3} + q_{x_{i_4}} = L_{i_4},\\
L'_{i_2} &= q_{x_1} + \ldots + q_{x_{i_1}} + q_{x_{i_4}} + \ldots + q_{x_{i_5}} + q_{x_{i_2}}\\
&= L_{i_5} + q_{x_{i_2}} = L_{i_1} + q_{x_{i_2}} = L_{i_2},\\
L'_{i_6} &= q_{x_1} + \ldots + q_{x_{i_1}} + q_{x_{i_4}} + \ldots + q_{x_{i_5}} + q_{x_{i_2}} + \ldots + q_{x_{i_3}} + q_{x_{i_6}}\\
&= L_{i_3} + q_{x_{i_6}} = L_{i_5} + q_{x_{i_6}} = L_{i_6}.
\end{aligned}$$

■

For the double–bridge move, assume that we choose from the tour four non–consecutive customers $x_{i_1}, x_{i_3}, x_{i_5}, x_{i_7}$, and that $x_{i_2}, x_{i_4}, x_{i_6}, x_{i_8}$ are their successors on the tour, respectively (see Figure 1).

Property 3.4. *If $L_{i_1} = L_{i_5}$ and $L_{i_3} = L_{i_7}$, then a new solution obtained by the double bridge-move is feasible.*

Proof. Assume, without loss of generality, that the depot is located between customers x_{i_8} and x_{i_1}. If the vehicle loads satisfy $L_{i_1} = L_{i_5}$ and $L_{i_3} = L_{i_7}$, a double–bridge modification can be performed without compromising feasibility. For instance, the loads at vertices x_{i_6}, x_{i_4}, x_{i_2}, and x_{i_8} remain unchanged.

$$\begin{aligned}
L'_{i_6} &= q_{x_1} + \cdots + q_{x_{i_1}} + q_{x_{i_6}} = L_{i_1} + q_{x_{i_6}} = L_{i_5} + q_{x_{i_6}} = L_{i_6},\\
L'_{i_4} &= q_{x_1} + \cdots + q_{x_{i_1}} + q_{x_{i_6}} + \cdots + q_{x_{i_7}} + q_{x_{i_4}}\\
&= L_{i_7} + q_{x_{i_4}} = L_{i_3} + q_{x_{i_4}} = L_{i_4},\\
L'_{i_2} &= q_{x_1} + \cdots + q_{x_{i_1}} + q_{x_{i_6}} + \cdots + q_{x_{i_7}} + q_{x_{i_4}} + \cdots + q_{x_{i_5}} + q_{x_{i_2}}\\
&= L_{i_5} + q_{x_{i_2}} = L_{i_1} + q_{x_{i_2}} = L_{i_2},\\
L'_{i_8} &= q_{x_1} + \cdots + q_{x_{i_1}} + q_{x_{i_6}} + \cdots + q_{x_{i_7}} + q_{x_{i_4}} + \cdots + q_{x_{i_5}} + q_{x_{i_2}} + \ldots q_{x_{i_3}} + q_{x_{i_8}}\\
&= L_{i_3} + q_{x_{i_8}} = L_{i_7} + q_{x_{i_8}} = L_{i_8}.
\end{aligned}$$

■

Therefore, we prune our shaking method by choosing four customers such that $L_{i_1} = L_{i_5}$ and $L_{i_3} = L_{i_7}$, this way preserving feasibility.

3.4.4. Sequential and Mixed General VNS for 1-PDTSP. Two new heuristics for the 1-PDTSP are summarized, as described in Algorithm 10. The first, Seq-GVNS, uses Seq-VND (Algorithm 6) as its local search routine. The second, Mix-GVNS, uses Mix-VND. Both heuristics rely on three neighborhoods: 2–opt, forward insertion, and backward insertion.

For the 2–opt and insertion moves, we use a tree data structure for efficient feasibility checks and updates (see Property 3.1 and Property 3.2). Feasibility after perturbation is ensured via random 3-opt and double-bridge moves (see Properties 3.3–3.4).

The k-th shaking step involves k repetitions ($k = 1, 2, \ldots, k_{\max}$) of random 3–opt* or double–bridge moves designed to preserve feasibility (see Figure 1). After performing local search, a move is accepted only if it results in an improved solution.

For both GVNS methods, two parameters need to be set by the user:

- $t_{\max}$-the total running time of the heuristic,
- $k_{\max}$-the total number of neighborhoods used in the outer loop (Shaking step).

Additionally, note that $\ell_{\max}$ the total number of local search routines used in Seq–VND), is equal to 3.

3.5. Computational results. This section presents the computational results of the proposed GVNS methods compared to existing algorithms from the literature. The experiments were performed on an Intel Core 2 Duo T5800 (2.0 GHz, 2 GB RAM, SPECint 2006=21.4) running Linux, using C++implementations compiled with the optimization flag `-O2`. Memory usage is minimal, at $\mathcal{O}(n^2)$, staying below 1 MB for the largest cases. Parallel processing was not utilized.

Different methods compared in this section were coded at different computers. Here is the list of methods and performances of the used computers:

- UB2–heuristics from [17]. The heuristic algorithms have been implemented in ANSI C, and ran on a personal computer AMD 1.33 GHz, under Windows 98 (SPECint 2006=12.6);
- Hybrid GRASP/VND metaheuristic algorithm from [18]. The algorithm was implemented in C++ and the program run on a personal computer with Intel Core 2 CPU at 2.4 GHz under Windows XP (SPECint 2006=28.1);
- Genetic algorithm from [40]. The algorithm is implemented in C under the Visual C++.net environment and run on a Pentium M 1.6 GHz PC with 256 MB of RAM (SPECint 2006=15.2.).

3.5.1. Test instances. All experiments were performed on the benchmark instances of the 1-PDTSP introduced in [18], available at http://webpages.ull.es/users/hhperez/PDsite/#XM94. These benchmark instances, for $n \leqslant 500$, are generated randomly as follows: $n-1$ customers are placed randomly within the square $[-500, 500] \times [-500, 500]$, each with a demand q_i ($2 \leqslant i \leqslant n$) randomly selected from the interval $[-10, 10]$. The depot is located at the origin $(0, 0)$ with a demand $q_1 = -\sum_{i=2}^{n} q_i$.

The travel cost $c(i, j)$ is calculated as the rounded Euclidean distance $d(i, j)$ between points i and j ($c(i, j) = \lfloor d(i, j) \rfloor$). Vehicle capacities are chosen from the set $\{10, 20, 40\}$. For each combination of n and Q, 10 random instances are generated and labeled with letters A, B, ..., I. In order to save space, only the average results across the 10 instances are reported.

The instances can be categorized into two groups: small instances, where n is chosen from $\{20, 30, 40, 50, 60\}$, and large instances, where n is selected from $\{100, 200, 300, 400, 500\}$. Note that for a given number of customers n, smaller values of Q result in more challenging problems. On the other hand, if Q is very large (e.g., $Q = 5000$, the 1–PDTSP effectively reduces to the TSP. Additionally, we generated instances with $n = 1000$ customers, allowing us to test our code on problems that are twice as large as the largest instances previously studied.

The optimal solutions for all small instances are known, as they were determined using the branch-and-cut algorithm described in [17]. For large instances, the optimal solutions are not available. However, the best–known results are reported in [18] and [40].

3.5.2. Main computational results. Tables 1 and 2 present GVNS results for small and large instances, respectively. For each $Q \in \{10, 20, 40\}$, we report the best and average vehicle tour lengths and the execution times (in seconds) to achieve the best value. The stopping condition $t_{\max} = n$ seconds is applied (while parameter $k_{\max}$ was set to 8). For small instances ($n \leqslant 60$), optimal tour lengths are reported.

CPU times for GA are missing in Tables 1 and 2, as [40] did not report them, nor are objective values provided for $Q \geqslant 20$. However, the authors claim that their GA method is faster than the hybrid heuristic in [18].

TABLE 1. % deviation from optimal values and running times on small 1-PDTSP benchmark instances of SEQ-GVNS, MIX-GVNS (suggested in this paper), GRASP/VND [18] and GA [40].

Parameters		Opt. value	Seq-GVNS		Mix-GVNS		GRASP/VND		GA		Running time (in seconds)		
n	Q	Av. in 10	Best	Av.	Best	Av.	Best	Av.	Best	Av.	SEQ-GVNS	MIX-GVNS	GRASP/VND
20	10	5402.50	0.00	0.38	0.02	0.68	0.00	0.00	0.00	0.00	0.03	0.47	0.07
	20	4237.30	0.00	0.10	0.00	0.10	0.00	0.00			0.01	0.10	0.03
	40	3977.40	0.00	0.37	0.00	0.28	0.00	0.00			0.01	0.04	0.02
30	10	6576.10	0.00	0.03	0.00	0.00	0.00	0.01	0.00	0.00	0.14	0.70	0.33
	20	5017.20	0.00	0.00	0.00	0.00	0.00	0.00			0.04	0.11	0.08
	40	4604.50	0.00	0.00	0.00	0.00	0.00	0.00			0.01	0.04	0.05
40	10	7189.30	0.00	0.07	0.00	0.00	0.00	0.40	0.00	0.04	0.78	1.95	0.80
	20	5500.10	0.00	0.00	0.00	0.00	0.00	0.00			0.10	0.15	0.15
	40	5142.50	0.00	0.00	0.00	0.00	0.00	0.00			0.03	0.18	0.09
50	10	8681.70	0.00	0.58	0.00	0.06	0.01	1.14	0.00	0.39	1.37	4.06	1.51
	20	6537.70	0.00	0.09	0.00	0.00	0.00	0.16			0.48	1.29	0.34
	40	5945.90	0.00	0.01	0.00	0.00	0.00	0.00			0.14	0.83	0.17
60	10	9165.80	0.00	0.70	0.00	0.11	0.15	2.03	0.00	0.74	2.22	7.02	2.40
	20	6956.00	0.03	0.12	0.00	0.00	0.08	0.34			0.73	2.81	0.49
	40	6361.70	0.00	0.00	0.00	0.00	0.00	0.00			0.26	0.88	0.26
Average		6086.38	0.00	0.16	0.00	0.08	0.02	0.27	0.00	0.23	0.42	1.38	0.45

In this study, we also compare our VNS with

- heuristics from Hernández–Pérez and Salazar-González [17] (only the best solution found);
- the hybrid GRASP/VND metaheuristic algorithm from Hernández–Pérez et al. [18] (the best solution, average solution and execution time);

TABLE 2. % deviation from best known values and running times on large 1-PDTSP benchmark instances of SEQ-GVNS, MIX-GVNS (suggested in this paper), GRASP/VND [18] and GA [40].

Parameters		Best known	Seq-GVNS		Mix-GVNS		GRASP/VND		GA		Running time (in seconds)		
n	Q	Av. in 10	Best	Av.	Best	Av.	Best	Av.	Best	Av.	SEQ-GVNS	MIX-GVNS	GRASP/VND
100	10	12718.60	0.29	1.52	0.00	0.35	1.88	4.62	0.96	1.85	9.96	23.79	8.85
	20	9357.60	0.02	0.96	0.00	0.18	0.65	2.55			5.89	14.95	2.22
	40	8165.40	0.00	0.20	0.00	0.04	0.00	0.57			1.84	5.97	0.69
200	10	18578.80	0.42	2.21	0.00	1.05	4.81	7.50	2.70	4.09	49.74	75.69	41.77
	20	13319.00	0.50	2.32	0.00	0.91	4.58	6.86			37.60	67.58	17.37
	40	11214.80	0.08	0.97	0.00	0.31	1.34	3.18			18.00	42.32	4.35
300	10	22935.30	0.83	2.39	0.00	1.47	5.30	7.67	4.20	5.62	104.61	122.83	117.86
	20	16313.40	0.88	2.60	0.00	1.43	6.60	8.62			38.74	115.93	50.90
	40	13671.40	0.41	1.40	0.00	0.56	2.85	4.80			24.91	88.63	12.89
400	10	25467.20	0.27	1.94	0.00	1.12	5.66	7.68	4.02	5.82	165.95	165.44	220.40
	20	18407.00	0.61	2.06	0.00	1.30	6.49	8.61			69.41	152.36	91.73
	40	15602.90	0.49	1.28	0.00	0.70	3.41	5.20			48.67	144.29	23.92
500	10	28774.20	0.00	1.54	0.10	1.28	5.80	7.76	5.57	7.37	124.14	209.76	391.01
	20	20927.00	0.17	1.70	0.00	1.38	6.53	8.43			107.01	194.76	164.77
	40	17495.50	0.41	1.50	0.00	0.86	4.47	6.10			89.81	193.52	43.98
Average		16863.21	0.36	1.64	0.01	0.86	4.04	6.01	3.49	4.95	59.75	107.85	79.51
1000	10	44744.20	0.96	19.74	0.00	1.52	-	-			349.59	393.22	-
	20	31661.10	1.64	3.57	0.00	1.56	7.69	8.95			478.08	441.03	618.33
	40	25450.00	1.20	2.66	0.00	1.28	6.64	8.14			474.72	430.16	440.00
Average of all		19.711.30	0.51	2.81	0.01	0.96	4.39	6.31	3.49	4.95	122.15	160.12	132.41

- the genetic algorithm from Zhao et al. [40] (best solution, average solution and standard deviation).

Clearly, our approach improves the best known solutions in all large benchmark instances, as showed in Table 2. For $Q = 10$ the hybrid GRASP/VND metaheuristic has failed to find a feasible solution, while in all other cases our VNS finds better solutions (5 percent on average).

Based on these results, we conclude that:

(1) the VNS based heuristic improves the best known solutions in all benchmark instances.
(2) VNS is robust. For some problem sizes, the worst average solution (out of 20) obtained by VNS is of better quality than the best ones obtained by other two methods.
(3) GA performs better than GRASP/VND, but instances with larger Q are not tested;
(4) using our VNS approach, we are able to efficiently solve instances with $n = 1000$ customers.

3.6. Conclusions. A General Variable Neighborhood Search (GVNS) based heuristics for solving the One-commodity pickup-and-delivery travelling talesman problem (1-PDTSP) is dveloped. This problem contains two NP-hard problems: TSP and the feasibility checking of the tour.

GVNS is a variant of VNS which consists of the usual Shaking operator but with several neighborhoods used in a deterministic fashion in the local search step.

For performing such a local search, known as the Variable Neighborhood Descent (VND), proposed the use of three different operators: 2–opt, insertion (forward and backward) and 3–opt. Moreover, these structures are used in two ways: sequentially (following the prescribed order) and nested. In order to store and update the incumbent solution, the segment tree structure is implemented and proved several properties regarding the complexity of the feasibility checking step.

The Shaking operator, uses 3–opt and double–bridge moves, maintaining the feasibility of the incumbent solution. Based on computational results, our method significantly improves the best-known solutions for all large instances in the literature. The average improvement over the previous state-of-the-art heuristics is up to 7%. For new large instances with $n = 1000$, the improvements reported are even bigger. These results are obtained in less computing time than the other methods, while keeping memory usage reasonable.

4. Traveling Deliveryman Problem

4.1. Introduction. The Traveling Deliveryman Problem (TDP), also known as the Minimum Latency Problem or Traveling Repairman Problem, differs from the TSP by aiming to minimize customers' total waiting time instead of the total tour length.

The Traveling Deliveryman Problem (TDP) is NP-hard in metric spaces (Blum et al. [4]). However, there are specific graph structures where TDP can be solved in polynomial time, such as paths and trees with diameters equal to 3 (Blum et al. [4]).

Wu et al. [39] proposed two exact methods for general graphs: Dynamic Programming and Branch and Bound (B&B). Their B&B solved instances with up to 50 customers in under 100 seconds but was less efficient than the CPLEX solver's MIP-based B&B.

Méndez-Díaz et al. [27] proposed an integer linear programming model and a B&B method for the Minimum Latency Problem, solving instances with up to $n = 40$ customers faster than the CPLEX solver. Abeledo et al. [1] proposed a branch-and-cut-and-price algorithm for TDP, solving TSPLIB instances with up to $n = 107$ customers exactly.

Salehipour et al. [35] introduced a heuristic combining GRASP with Variable Neighborhood Descent (VND), a deterministic version of VNS. Their sequential VND uses five neighborhood structures: (i) adjacent swaps (1-opt), (ii) swaps, (iii) 2-opt, (iv) Or-opt, and (v) drop/add moves.

We propose a General VNS for the TDP, incorporating neighborhood structures from similar problems into the VNS framework for the first time. Preprocessing and intelligent updates improve efficiency, with neighborhood moves matching the TSP's worst-case complexity. Results on standard instances show that our heuristic outperforms recent methods, setting a new state-of-the-art for the TDP.

4.2. Problem formulation.

Combinatorial formulation. Let $G = (V, A)$ be a complete, directed, and asymmetric graph, where $V = 0, 1, \ldots, n$ is a set of nodes. Nodes $1, 2, \ldots, n$ are delivery nodes, and node 0 is the depot. The distance $d(i, j)$ is associated with each arc

$(i, j) \in A$. The Traveling Deliveryman Problem (TDP) consists of determining a Hamiltonian path on the graph:

$$x = (x_0, x_1, x_2, x_3, \ldots, x_n),$$

starting from the depot node $x_0 = 0$, with the objective of minimizing the total waiting time for all customers. Assume the deliveryman travels at a constant velocity v. The time required to reach customer i is the distance traveled from the depot to customer i divided by the velocity v.

The distance to reach the first client, x_1, is $\delta_1 = d(x_0, x_1)$. Similarly, to reach the second client, x_2, the distance traveled is $\delta_2 = d(x_0, x_1) + d(x_1, x_2)$. In general, to reach client x_k, the total distance traveled is

$$\delta_k = d(x_0, x_1) + d(x_1, x_2) + d(x_2, x_3) + \cdots + d(x_{k-1}, x_k).$$

Therefore, the corresponding time of reaching the customer x_k is as follows:

$$t_k = \frac{\delta_k}{v}.$$

Since the objective function represents the sum of all such obtained times (for all k), it can be formulated as

$$\sum_{k=1}^{n} t_k.$$

The last expression may be formulated more simply as

$$\sum_{k=1}^{n} \frac{(n-k+1)d(x_{k-1}, x_k)}{v}.$$

Since v is constant, its value does not influence the optimal solution. Thus, the objective function to be minimized may be presented as

$$f(x) = \sum_{k=1}^{n} (n-k+1)d(x_{k-1}, x_k). \tag{4.1}$$

Note that the time needed to go back to the depot is not included in the objective function, since there is no customer waiting at that time.

4.2.1. Mixed integer programming (MIP) formulation. As for the TSP, the TDP can also be formulated in different ways. Fischetti et al. (1993) used the so-called flow MIP formulation. The two sets of variables are defined as

$$X_{ij} = \begin{cases} 1, & \text{if deliveryman travels arc}(i, j) \\ 0, & \text{otherwise} \end{cases}$$

$$Y_{ij} = \begin{cases} 0, & \text{if arc } (i, j) \text{ is not used} \\ n-k, & \text{if } (i, j) \text{ is } k\text{-th arc on route} \end{cases}$$

Then the MIP formulation for the TDP is as follows:

$$\min \sum_{i=0}^{n} \sum_{j=0}^{n} d(i, j) Y_{ij},$$

subject to

$$\sum_{j=0,j\neq i}^{n} X_{ij} = 1, \quad i = 0,1,2,\ldots,n \tag{4.2}$$

$$\sum_{i=0,i\neq j}^{n} X_{ij} = 1, \quad j = 0,1,2,\ldots,n \tag{4.3}$$

$$\sum_{i=1}^{n} Y_{0i} = n \tag{4.4}$$

$$\sum_{i=0,i\neq k}^{n} Y_{ik} - \sum_{i=0,i\neq k}^{n} Y_{ki} = 1 \tag{4.5}$$

$$Y_{i0} = 0 \quad i = 1,2,\ldots,n \tag{4.6}$$

$$Y_{0i} \leqslant nX_{0i}, \quad i = 1,2,\ldots,n$$

$$Y_{ij} \leqslant (n-1)X_{ij}m \quad i,j = 1,2,3,\ldots,n;\ i \neq j \tag{4.7}$$

$$X_{ij} \in \{0,1\}, \quad i,j = 0,1,2,\ldots,n;\ i \neq j$$

$$Y_{ij} \in \{0,\ldots,n\}, \quad i,j = 0,1,2,\ldots,n;\ i \neq j$$

In this formulation, there are $2n(n+1)$ variables and n^2+4n+3 constraints. Constraints (4.2) and (4.3) ensure that each customer is visited and departed exactly once, meaning each customer has one incoming and one outgoing arc. Constraints (4.4)–(4.5) are flow constraints, which prevent the final tour X from containing subtours (i.e., tours that exclude any of the n clients). Constraints (4.6)–(4.7) establish a logical relationship between the variables X and Y. For instance, if $X_{ij} = 0$, then Y_{ij} must also be 0.

4.2.2. Lower bound. Obviously, a minimum spanning tree (MST) of the corresponding weighted graph, may be used as a lower bounding procedure. Let us rank edges $e \in A$ of the MST in a nondecreasing order of their weights

$$d(e_1) \leqslant d(e_2) \leqslant \cdots \leqslant d(e_n).$$

Then the lower bound LB is given by

$$LB = \sum_{k=1}^{n} (n+1-k)d(e_k). \tag{4.8}$$

This formula proposed for the first by Salehipour et al. [35].

Usual local search procedures used for solving TSP are 2-opt, Or-opt (or $1-2-3$- insertion) and 3-opt. Their advantage is in the fact that the updating of the objective function may be obtained in constant time. However, this is not the case when solving TDP, at least if usual data structure is used and no preprocessing step is performed. We first show that the complexity of k-opt moves in such cases is $\mathcal{O}(n^{k+1})$ (instead of $\mathcal{O}(n^k)$ for solving classical TSP). Then we explain the steps necessary to be done, so as the data structure to be used in order to get the same complexity for TDP as it is for TSP, i.e., $\mathcal{O}(n^k)$.

We first demonstrate that the complexity of k-opt moves for TDP in such cases is $\mathcal{O}(n^{k+1})$, compared to $\mathcal{O}(n^k)$ for the classical TSP. Then, we outline the necessary steps and modifications to the data structure to achieve the same complexity for TDP as for TSP, namely $\mathcal{O}(n^k)$ (for $k \in \{1, 2, 3\}$).

4.3. k-opt complexity. Let us consider the solution x of TDP

$$x = (x_0, x_1, \ldots, x_{i-1}, x_i, x_{i+1}, \ldots, x_{j-1}, x_j, x_{j+1} \ldots, x_n)$$

The solution that belongs to 2-opt neighborhood of the solution x (in solving TSP) is obtained after deleting the links between x_i and x_{i+1}, and between x_j and x_{j+1}. The new tour x' and its change of the objective function value $f(x') - f(x)$ are as follows:

$$x' = (x_0, x_1, \ldots, x_{i-1}, x_i, x_j, x_{j-1}, \ldots, x_{i+2}, x_{i+1}, x_{j+1} \ldots, x_n) \tag{4.9}$$

$$\Delta f = f(x') - f(x) = d(x_i, x_j) + d(x_{i+1}, x_{j+1}) - d(x_i, x_{i+1}) - d(x_j, x_{j+1}) \tag{4.10}$$

The function $f(x')$ is clearly computed with just four additions, making it a constant-time operation. However, this efficiency does not extend to solving the TDP using the 2-opt method. In a straightforward 2-opt implementation for TDP, updating $f(x')$ requires modifying all coefficients between x_{i+1} and x_j in formula (4.1). As a result, calculating the difference Δf involves $\mathcal{O}(j - i)$ operations. In the worst-case scenario, where $j = n$ and $i = 1$, Δf is computed in $\mathcal{O}(n)$.

Furthermore, note that this basic implementation increases the complexity of a k-opt move for TDP by a factor of n compared to its application in TSP. Specifically, the complexity becomes $\mathcal{O}(n^{k+1})$. However, in this section, we shall demonstrate that by employing an appropriate data structure, the worst-case complexity of k-opt for solving both TDP and TSP can be reduced to the same level: $O(n^k)$.

Our list of neighborhoods consists of (i) 1-opt, (ii) move forward (insertion to the right); (iii) move backward (insertion to the left); (iv) 2-opt. We did not put 3-opt move in our list because of its large complexity $O(n^3)$.

4.3.1. 1-opt. 1-opt move is a special case of 2-opt, where the index j is set to $i+2$ in (4.9). Let us consider four consecutive clients in the current tour x:

$$x_k, x_{k+1}, x_{k+2}, x_{k+3}. \tag{4.11}$$

Then the new tour x' is obtained by swapping x_{k+1} and x_{k+2}.

Property 4.1. *Exploring complete* 1*-opt neighborhood in solving TDP is in* $\mathcal{O}(n)$.

Proof. From (4.1), (4.9) and (4.11) we see that the difference Δf between the new objective function and the old one is

$$\Delta f = (n-k)(d(x_k, x_{k+2}) - d(x_k, x_{k+1})) + (n-k-2)(d(x_{k+1}, x_{k+3}) - d(x_{k+2}, x_{k+3})).$$

If Δf is negative, then a better tour is obtained. Clearly, finding Δf is in $\mathcal{O}(1)$. Since the cardinality of 1-opt neighborhood is $n - 3$, the result follows. ∎

4.3.2. Move forward (mf). The solution from this neighborhood is defined by moving ℓ consecutive customers to the end of the tour ($\ell = 1, 2, \ldots, \ell_{\max}$). If that block of ℓ customers starts from the position $i+1$ and is inserted after the position j, (i.e., after customer x_j), then the tour

$$x = (x_1, x_2, \ldots, x_i, x_{i+1}, \ldots, x_j, x_{j+1}, \ldots, x_n)$$

becomes

$$x' = (x_1, \ldots, x_i, x_{i+\ell+1}, \ldots, x_j, x_{i+1}, \ldots, x_{i+\ell}, x_{j+1}, \ldots, x_n).$$

Property 4.2. *Exploring complete move-forward neighborhood in solving TDP is in* $\mathcal{O}(n^2\ell_{\max})$.

Proof. The difference between the new and the old objective functions is

$$\begin{aligned}\Delta f = \ell\delta(x_{i+\ell+1}, x_j) - (j-i-\ell)\delta(x_{i+1}, x_{i+\ell}) \\ + (n-i)(d(x_i, x_{i+\ell+1}) - d(x_i, x_{i+1})) \\ + (n-j)(d(x_{i+\ell}, x_{j+1}) - d(x_j, x_{j+1})) \\ + (n-j+\ell)d(x_j, x_{i+1}) - (n-i-\ell)d(x_{i+\ell}, x_{i+\ell+1})\end{aligned}$$

where $\delta(x_{i+\ell+1}, x_j)$ denotes a subtour distance from the customer $x_{i+\ell+1}$ to the customer x_j. Similarly, $\delta(x_{i+1}, x_{i+\ell})$ denotes the distance from the client's position $i+1$ to the position $i+\ell$, i.e.

$$\delta(x_i, x_j) = \sum_{h=i}^{j-1} d(x_h, x_{h+1})$$

If i and j are fixed, then increase of ℓ by 1 makes calculation of the new Δf in $\mathcal{O}(1)$. The same holds for all (i, j) pairs. Therefore, finding objective function values in complete move-forward neighborhood is $\mathcal{O}(\binom{n}{2}\ell_{\max} = O(n^2\ell_{\max})$. ■

4.3.3. Move backward (mb). This neighborhood is similar to the previous one, with the key difference being that a block of ℓ consecutive customers is shifted to the beginning of the tour. These two neighborhoods can be considered as a single neighborhood, referred to as the ℓ-insertion neighborhood. As such, they represent an extension of the Or-opt move, where $\ell = 1, 2$, or 3. However, we decide to divide the ℓ-insertion neighborhood into two distinct neighborhoods. This approach has proven effective for solving certain other graph optimization problems (see [5]).

4.3.4. 2–opt move. We have previously described the 2-opt move (refer to Eqs. (4.9) and (4.10)). Here, we show that the change in the objective function value of 2-opt move can be performed in $\mathcal{O}(1)$ time if a suitable data structure is employed for storing and updating the current incumbent tour.

Property 4.3. *Exploring complete* 2*-opt neighborhood in solving TDP is in* $\mathcal{O}(n^2)$.

Proof. Let the incumbent tour be presented as

$$x = (x_1, x_2, \ldots, x_{i-1}, x_i, x_{i+1}, \ldots, x_{j-1}, x_j, x_{j+1}, \ldots, x_n).$$

After deleting links (x_i, x_{i+1}) and (x_j, x_{j+1}), the new tour x' becomes

$$x' = (x_1, x_2, \ldots, x_{i-1}, x_i, x_j, x_{j-1}, \ldots, x_{i+2}, x_{i+1}, x_{j+1}, \ldots, x_n).$$

The corresponding objective function values are

$$\begin{aligned} f(x) = {} & (n-1)d(x_1, x_2) + \cdots + (n-i+1)d(x_{i-1}, x_i) + (n-i)d(x_i, x_{i+1}) \\ & + (n-i-1)d(x_{i+1}, x_{i+2}) + \cdots + (n-j+1)d(x_{j-1}, x_j) \\ & + (n-j)d(x_j, x_{j+1}) + (n-j-1)d(x_{j+1}, x_{j+2}) + \cdots + d(x_{n-1}, x_n) \end{aligned}$$

and

$$\begin{aligned} f(x') = {} & (n-1)d(x_1, x_2) + \cdots + (n-i+1)d(x_{i-1}, x_i) + (n-i)d(x_i, x_j) \\ & + (n-i-1)d(x_j, x_{j-1}) + \cdots + (n-j+1)d(x_{i+2}, x_{i+1}) \\ & + (n-j)d(x_{i+1}, x_{j+1}) + (n-j-1)d(x_{j+1}, x_{j+2}) + \cdots + d(x_{n-1}, x_n) \end{aligned}$$

After summing these two equations we get

$$\begin{aligned} f(x) + f(x') = {} & 2\sum_{k=1}^{i-1}(n-i)d(x_i, x_{i+1}) + 2\sum_{k=j+1}^{n-1}(n-i)d(x_i, x_{i+1}) \\ & + (2n-i-j)\delta(x_{i+1}, x_{j-1}) + (n-i)(d(x_i, x_{i+1}) + d(x_i, x_j)) \\ & + (n-j)(d(x_{i+1}, x_{j+1}) + d(x_j, x_{j+1})) \end{aligned}$$

By selecting suitable data structures and performing preprocessing (prior to exploring the neighborhood), the last formula can be computed in $\mathcal{O}(1)$ time. Additionally, this process simultaneously determines whether x' is an improved solution. Since $f(x)$ is already known, $f(x')$ can also be calculated efficiently.

By choosing appropriate data structures and preprocessing, the last formula can be computed in $\mathcal{O}(1)$ time. This also allows for simultaneous determination of whether x' is an improved solution, as $f(x')$ can be efficiently calculated once $f(x)$ is known.

The challenge lies in calculating the formula mentioned above. Therefore, before beginning the search through the 2-opt neighborhood, it is necessary to compute and store two sequences, $S_b(i)$ and $S_e(i)$, which represent the following sums:

$$S_b(i) = \sum_{k=1}^{i}(n-k)d(x_k, x_{k+1}) \quad \text{and} \quad S_e(i) = \sum_{k=i}^{n-1}(n-k)d(x_k, x_{k+1})$$

so as

$$\delta(i) = \sum_{k=1}^{i} d(x_k, x_{k+1}).$$

All can be done in $O(n)$ time. If we have these values, then

$$\begin{aligned} f(x) + f(x') = {} & 2S_b(i-1) + 2S_e(j+1) + (2n-i-j)(\delta(j-2) - \delta(i)) \\ & + (n-i)(d(x_i, x_{i+1}) + d(x_i, x_j)) + (n-j)(d(x_{i+1}, x_{j+1}) + d(x_j, x_{j+1})). \end{aligned}$$

clearly, this value can be computed in $\mathcal{O}(1)$ time. Consequently, the entire 2-opt neighborhood can be explored in $\mathcal{O}(n^2)$. ■

4.3.5. Double bridge (db). Double bridge is a special case of 4–opt. For any four indices i, j, k and ℓ $(i < j < k < \ell)$, edges (x_i, x_{i+1}), (x_j, x_{j+1}), (x_k, x_{k+1}), $(x_\ell, x_{\ell+1})$ are removed. New 4 edges are then $(x_i, x_{k+1}), (x_\ell, x_{j+1})$, (x_k, x_{i+1}) and $(x_j, x_{\ell+1})$. It is very important for some combinatorial problems since it keeps the orientation of the tour. The neighborhood size of double bridge is $\mathcal{O}(n^4)$. However, we reduce it to $\mathcal{O}(n^2)$ by fixing indices $j = i + 2$ and $\ell = k + 2$. The double-bridge move is used only within our GVNS-2, which will be explained later in Algorithm 6.

4.3.6. Interchange (Exchg). When two non-consecutive customers, x_i and x_j $(i < j)$, exchange their positions in the tour, the objective function value naturally changes. This interchange move is a specific case of the 4-opt move. The size of the entire neighborhood is $\mathcal{O}(n^2)$. We use this move solely to compare various multi-start local search procedures with the multi-start VND.

4.4. General VNS for solving TDP. General VNS (GVNS) is a variant of VNS where VND is used as a local search routine within basic VNS scheme [13, 14]. It contains 3 parameters: (i) $t_{\max}$ – maximum time allowed in a search; (ii) $k_{\max}$ – the number of neighborhoods used in shaking; (iii) $\ell_{\max}$ – the number of neighborhoods used in VND. Its pseudo code is given in Algorithm 13.

Algorithm 13: Steps of the general VNS for TDP.

```
Function GVNS (x, ℓ_max, k_max, t_max);
x ←InitialSolution;
repeat
    k ← 1;
    repeat
        x′ ← Shake(x, k);
        x″ ← VND(x′, ℓ_max);
        if f(x″) < f(x) then
            x ← x″;
            k ← 1;
        else
            k ← k + 1;
        end
    until k > k_max;
until t > t_max;
return x;
```

4.5. Initial solution. We employ two methods to obtain an initial solution. The first method is a random approach, where the solution is generated as a random permutation of the $n - 1$ customers. The second method is a Greedy Randomized procedure, detailed in Algorithm 14, which we refer to as Greedy-R.

In each step of the Greedy-R(x, q) procedure, the next customer x_k is randomly selected from the q closest customers to the previously selected customer x_{k-1},

Algorithm 14: Greedy randomized algorithm for initial solution.

Function GREEDY-R(x, q);
$S \leftarrow \{1, 2, 3, \ldots, n\}$;
$x_0 \leftarrow 0$;
$k \leftarrow 1$;
while $S \neq \emptyset$ **do**
  $S' \leftarrow$ subset of S with the $\min(q, n-k)$ customers nearest to x_{k-1};
  $x_k \leftarrow$ Random selected vertex from S';
  $S \leftarrow S \setminus \{x_k\}$;
  $k \leftarrow k + 1$;
end
return x

where q is a predefined parameter. If the total number of customers n or the number of remaining customers $n-k$ is less than q, one of the remaining customers is selected at random. In our implementation, the parameter q is consistently set to 10 ($q = 10$).

4.6. Shaking. In the VNS shaking step, the current solution x is perturbed to generate a random solution x' from the k-th neighborhood of x ($x' \in \mathcal{N}_k(x)$, $k = 1, \ldots, k_{\max}$). This random solution serves as the initial input for the subsequent local search routine. Based on experimental results, we chose the insertion neighborhood structure for the shaking process. Specifically, the shaking involves randomly selecting a customer i and inserting it between customers j and $j+1$, where j is also chosen randomly. This random insertion move is repeated k times.

Algorithm 15: Shaking procedure.

function SHAKE(x, k);
while $k > 0$ **do**
  $i \leftarrow$RANDOM$(1, n)$; //select i from $[1, n]$ at random;
  $j \leftarrow$RANDOM$(1, n)$; // select j from $[1, n]$ at random;
  if $i \neq j$ **then**
    Insert x_i between x_j and x_{j+1};
    $k \leftarrow k - 1$
  end
end
return x

It is important to note that an insertion move can involve either a forward or backward shift. Specifically, the shaking process corresponds to a forward move if $j \geqslant i$ and a backward move if $j \leqslant i$. Additionally, a single insertion move is a special case of the Or-opt move, where one, two, or three consecutive vertices are relocated.

4.7. VND algorithms for TDP. Determining the optimal order of neighborhoods in a deterministic VND heuristic can significantly impact the quality of the final solution. Following extensive testing of various configurations, we adopted the following order of neighborhoods: (i) 1-opt, (ii) 2-opt, (iii) move-forward, and (iv) move-backward. In Mix-VND, we also include (v) the double-bridge neighborhood structure.

To reduce the number of solutions visited in the 2-opt neighborhood, for a fixed i (i.e., x_i), we consider only r (a predefined parameter) nearest customers as candidates for x_j when performing the 2-opt move. If the total number of customers is less than r, we set $r = n/2$. To facilitate this, we rank all distances (columns of matrix D) during a pre-processing step, producing a ranked distance matrix R that allows us to efficiently identify the r closest customers for each x_i.

We developed two VND variants for solving the TDP: Seq-VND and Mix-VND (see Algorithms 16 and 17).

In Mix-VND we perform Seq-VND starting from each solution that belong to N_{db} neighborhood of a current solution x. The N_{db} neighborhood is fully explored but in the first improvement strategy. If the solution x'' obtained with Seq-VND heuristic is better than the current solution x', we move there ($x' \leftarrow x''$).

Algorithm 16: Sequential VND for solving TDP.

```
function Seq-VND(R, x', r);
ℓ ← 1;
while ℓ ⩽ 4 do
    if ℓ = 1 then
        x'' ← argmin{f(x)|x ∈ N_{1−opt}(x')};
    end
    if ℓ = 2 then
        x'' ← argmin{f(x)|x ∈ N_{2−opt}(R, x', r)};
    end
    if ℓ = 3 then
        x'' ← argmin{f(x)|x ∈ N_{mf}(x')};
    end
    if ℓ = 4 then
        x'' ← argmin{f(x)|x ∈ N_{mb}(x')};
    end
    if f(x'') < f(x') then
        x' ← x'';
        ℓ ← 1;
    else
        ℓ ← ℓ + 1;
    end
end
return x'
```

In Mix-VND, we apply Seq-VND starting from each solution within the N_{db} neighborhood of the current solution x. The N_{db} neighborhood is fully explored using a first-improvement strategy. If the solution x'' obtained through the Seq-VND heuristic is better than the current solution x', we update the current solution by setting $x' \leftarrow x''$.

Algorithm 17: Mixed Variable Neighborhood Descent for TDP.

```
function MIX-VND (R, x', r);
improve ← true;
while improve do
    improve ← false;
    while there are unexplored solution in N_db(x') and improve = false
     do
        x'' ← next solution in N_db(x');
        x'' ←SEQ-VND(R, x'', r);
        if f(x'') < f(x') then
            x' ← x'';
            improve ← true;
        end
    end
end
return x'
```

4.8. GVNS algorithms for TDP. We developed two GVNS variants for solving the TDP. The first variant uses sequential VND as the local search method (GVNS-1), while the second employs mixed nested VND (GVNS-2). Additionally, we implemented two approaches for generating the initial solution: Greedy-R$(x, 10)$ and Rand(x). Note that Rand is a procedure that generates a random permutation of n numbers and is not described in detail here.

4.9. Computational results for TDP. Our GVNS-1 and GVNS-2 implementations are written in C++. The computational analysis was conducted on an Intel Pentium IV processor running at 1.8 GHz, under a Linux operating system.

4.9.1. Test instances. Salehipour et al. [35] noted the lack of TDP-specific test instances in the literature. To address this, they created their own instances and used TSP library instances to evaluate their GRASP+VNS-based heuristics.

Salehipour et al. [35] generated a total of 140 instances with n taking values of $10, 20, 50, 100, 200, 500$, and 1000. For each n, 20 distinct instances were created as follows: the coordinates of all customers, including the depot, were randomly generated using a uniform distribution within the interval $[0, 100]$. For instances with 500 and 1000 customers, this interval was expanded to $[0, 500]$. The distance matrix $D = (d_{ij})$ was constructed by taking the floor of the Euclidean distance between any two nodes i and j.

Algorithm 18: First variant of GVNS for TDP (GVNS-1).

```
Function GVNS (D, x, k_max, t_max);
R ←Rank(D);
x ←Greedy-R(x, 10) (or x ←Rand(x));
repeat
    k ← 1;
    repeat
        x' ←Shake(x, k);
        x'' ← Seq − VND(R, x', 4);
        if f(x'') < f(x) then
            x ← x'';
            k ← 1;
        else
            k ← k + 1;
        end
    until k > k_max;
until t > t_max;
return x;
```

Algorithm 19: Second variant of GVNS for TDP (GVNS-2).

```
Function GVNS (D, x, k_max, t_max);
R ← Rank(D);
x ← Greedy − R(x, 10) (or x ← Rand(x));
repeat
    k ← 1;
    repeat
        x' ← Shake(x, k);
        x'' ← Mix − VND(R, x', 4);
        if f(x'') < f(x) then
            x ← x'';
            k ← 1;
        else
            k ← k + 1;
        end
    until k > k_max;
until t > t_max;
return x;
```

4.9.2. Stopping condition. Our VNS-based heuristic is terminated after n seconds ($t_{\max} = n$ seconds). In the tables below, we report the time at which the best solution for each instance is achieved.

4.9.3. Comparison. In this section, the average percentage deviation from the lower bound (LB, calculated by using (4.8)) is used to evaluate the quality of the heuristics, following the approach in [35]. The average deviation is computed across 20 instances generated for the same n.

Table 3 presents a comparison of four methods: GRASP+VND, GRASP+VNS, GVNS-1, and GVNS-2. For each method, the average deviation from the lower bound and the CPU time are reported. Columns 8 and 11 show the percentage improvement of the proposed GVNS-1 and GVNS-2 methods over GRASP+VNS, which is considered the current state-of-the-art heuristic.

Table 4 provides the same information as Table 3 but focuses on instances from the TSP library. Instead of the percentage deviation from the lower bound, the objective function values are reported, obtained from a single run of each method.

TABLE 3. Comparison on random instances.

n	GRASP+VND		GRASP+VNS		GVNS-1			GVNS-2		
	%dev.	time	%dev.	time	%dev.	time	impr.	%dev.	time	impr.
10	33.04	0.00	33.04	0.00	33.04	0.07	0.00	33.04	0.23	0.00
20	39.63	0.00	40.34	0.01	39.21	0.41	2.88	39.29	0.87	2.67
50	45.30	0.00	47.20	0.08	43.90	2.41	7.52	43.97	4.36	7.35
100	43.26	1.11	44.28	2.95	40.79	25.86	8.56	40.82	38.41	8.48
200	43.16	3.75	38.77	68.03	37.95	124.32	1.11	38.14	93.80	0.60
500	46.11	604.89	49.50	1524.09	39.41	414.05	25.60	39.54	421.72	25.19
1000	45.99	3876.51	42.35	9311.21	41.51	942.47	2.02	41.28	921.52	2.59

TABLE 4. Comparison on TSP-Lib instances.

Name	GRASP+VND		GVNS-1			GVNS-2		
	obj.val.	time	obj.val.	time	impr.	obj.val.	time	impr.
st70	19533	22.24	19215	7.6	1.63	19215	26.15	1.63
att532	18448435	65121.06	17610285	488.95	4.54	17584979	532.02	4.68
kroD100	976830	54.69	949594	16.42	2.79	949594	46.55	2.79
lin105	585823	101.27	585823	86.76	0	585823	76.17	0.00
lin318	5876537	3951.02	5612165	179.13	4.5	5601351	294.85	4.68
pr107	1983475	225.32	1980767	97.51	0.14	1981141	49.94	0.12
pr226	7226554	2328.71	7106546	144.29	1.66	7100708	37.57	1.74
pr439	18567170	4563.44	17790562	327.15	4.18	17799975	396.84	4.13
rat99	56994	195.31	54984	42.41	3.53	55028	50.59	3.45
rat195	213371	884.78	210193	111.2	1.49	210319	150.94	1.43

TABLE 5. Comparison of the results obtained by branch–and–cut–and–price method proposed by Abeledo et al. (2010) and by our GVNS-1.

Name	B&C&P		GVNS-1	
	obj.val.	time	obj.val.	time
dantzig42	**12528**	28	**12528**	0.25
swiss42	**22327**	20	**22327**	0.09
att48	**209320**	103	**209320**	0.07
gr48	**102378**	76	**102378**	0.
12 hk48	**247926**	124	**247926**	0.98
eil51	**10178**	33	**10178**	1.98
berlin52	**143721**	104	**143721**	1.38
brazil58	**512361**	633	**512361**	6.16
st70	**20557**	2895	**20557**	2.68
eil76	**17976**	223	**17976**	1.36
pr76	**3455242**	58045	**3455242**	27.77
gr96	**2097170**	160250	**2097170**	2.08
rat99	57986	¿48hs	57986	23.99
kroA100	**983128**	106336	**983128**	52.05
kroB100	**986008**	7684	**986008**	29.76
kroC100	**961324**	39559	**961324**	1.94
kroD100	976965	¿48hs	976965	6.97
kroE100	**971266**	117965	**971266**	2.90
rd100	**340047**	18582	**340047**	63.82
eil101	27513	¿48hs	27513	61.91
lin105	**603910**	6317	**603910**	12.50
pr107	**2026626**	9947	**2026626**	4.23

The results indicate that: (i) GVNS-1 and GVNS-2 outperform both GRASP+VND and GRASP+VNS; (ii) There is no significant difference between GVNS-1 and GVNS-2 overall. However, GVNS-2 tends to perform better for larger instances, whereas GVNS-1 shows better results for small and medium-sized instances.

Table 5 compares the results obtained using the branch-and-cut-and-price method (B & C & P) proposed by Abeledo et al. [1] with those achieved by our GVNS-1. Note that the objective function differs slightly in this case: the first edge in the tour is weighted by $n + 1$, the second by n, and so on, with the edge connecting the last customer on the tour to the depot weighted by 1. Values shown in bold indicate proven optimality. The results demonstrate that GVNS-1 achieves all optimal values and three best-known values with significantly lower CPU time.

4.10. Conclusions. The neighborhoods we propose for the Traveling Deliveryman Problem (TDP) are classical ones traditionally used to solve the Traveling Salesman Problem (TSP). We demonstrate how these neighborhoods can be efficiently

adapted for TDP and analyze the complexity of these adapted moves. Extensive computational results reveal that our two GVNS variants outperform the recent heuristic proposed by Salehipour et al. (2011). Future research could extend the methodology and data structures to other similar combinatorial optimization problems. Additionally, customer clustering on the network, as suggested by Carrizosa et al. (2011), could be considered as a preprocessing step before routing.

5. Conclusions

In this chapter, we first summarize the principles of Variable Neighborhood Search (VNS) in general terms. We then demonstrate how this methodology can be applied to two variants of the Traveling Salesman Problem. Specifically, neighborhoods used for the classical TSP are customized for the Pickup and Delivery TSP and Traveling Deliveryman Problem. This customization involves using new efficient data structures for solution representation, enabling the implementation of highly efficient variants of General VNS for these problems.

Acknowledgments. The research has been partially supported by the Serbian Ministry of Science, Technological Development, and Innovations, Agreement No. 451-03-136/2025-03/200029.

References

[1] H. Abeledo, R. Fukasawa, A. Pessoa, E. Uchoa, *The time dependent traveling salesman problem: polyhedra and algorithm*, Math. Program. Comput. **5** (2013), 27–55.

[2] Z. Amirgaliyeva, N. Mladenović, R. Todosijević, D. Urošević, *Solving the maximum min-sum dispersion by alternating formulations of two different problems*, Eur. J. Oper. Res. **260** (2017), 444–459.

[3] S. Anily, J. Brame, *Approximation algorithms for the capacitated travelling salesman problem with pickups and deliveries*, Nav. Res. Logist. **46** (1999), 654–670.

[4] A. Blum, P. Chalasani, D. Coppersmish, B. Pulleyblank, P. Raguavan, M. Sudan, *The minumum latency problem*, In: Proceedings of 26^{th} annual ACM symposium on theory of computting, Montreal, 1994.

[5] J. Brimberg, N. Mladenović, D. Urošević, *Variable neighborhood search for the heaviest k-subgraph*, Comput. Oper. Res. **36** (2009), 2885–2891.

[6] J. Brimberg, N. Mladenović, R. Todosijević, D. Urošević, *A basic variable neighborhood search heuristic for the uncapacitated multiple allocation p-hub center problem*, Optim. Lett. **11** (2017), 313–327.

[7] J. Brimberg, N. Mladenović, R. Todosijević, D. Urošević, *General variable neighborhood search for the uncapacitated single allocation p-hub center problem*, Optim. Lett. **11** (2017), 377–388.

[8] J. Brimberg, N. Mladenović, R. Todosijević, D. Urošević, *Solving the capacitated clustering problem with variable neighborhood search*, Ann. Oper. Res. **272** (2019), 289–321.

[9] E. Carrizosa, N. Mladenović, R. Todosijević, *Variable neighborhood search for minimum sum-of-squares clustering on networks*, Eur. J. Oper. Res. **230** (2013), 356–363.

[10] A. Duarte, J. Sánchez–Oro, N. Mladenović, R. Todosijević, *Variable Neighborhood Descent*, In: R. Martí, P. M. Pardalos, M. G. C. Resende (eds.), *Handbook of heuristics*, 341–367, Springer, Cham, 2018.

[11] P. M. Fenwick, *A new data structure for cumulative frequency tables*, Software–Practice and Experience **24** (1994), 327–336.

[12] P. Hansen, N. Mladenović, *First vs. best improvement: An empirical study*, Discrete Appl. Math. **154** (2006), 802–817.

[13] P. Hansen, N. Mladenović, J. A. Moreno-Pérez, *Variable neighbourhood search: Methods and applications*, 4OR **6** (2008), 319–360.
[14] P. Hansen, N. Mladenović, J. A. Moreno Pérez, *Variable neighbourhood search: algorithms and applications*, Ann. Oper. Res. **175** (2010), 367-–407.
[15] P. Hansen, N. Mladenović, R. Todosijević, S. Hanafi, *Variable neighborhood search: basics and variants*, EURO J. Comput. Optim. **5** (2017), 423–454.
[16] H. Hernández-Pérez, J. J. Salazar-González, *Heuristics for the one-commodity pickup-and-delivery travelling salesman problem*, Transp. Sci. **38** (2004), 245–255.
[17] H. Hernández-Pérez, J. J. Salazar-González, *A branch-and-cut algorithm for a travelling salesman problem with pickup and delivery*, Discrete Appl. Math. **145** (2004), 126–139.
[18] H. Hernández-Pérez, I. Rodríguez-Martín, J. J. Salazar-González, *A hybrid GRASP/VND heuristic for the one-commodity pickup-and-delivery travelling salesman problem*, Comput. Oper. Res. **36** (2009), 1639–1645.
[19] A. Ilić, D. Urosević, J. Brimberg, N. Mladenović, *Variable neighborhood search for solving the uncapacitated single allocation p-hub median problem*, Eur. J. Oper. Res. **206** (2010), 289–300.
[20] S. Irnich, B. Funke T. Grunert, *Sequential search and its application to vehicle-routing problems*, Comput. Oper. Res. **33** (2006) 2405–2429
[21] S. Irnich, *A unified modeling and solution framework for vehicle routing and local search-based metaheuristics*, INFORMS J. Comput. **20** (2008), 270–287.
[22] D.S. Johnson, L.A. McGeoch, E.E. Rothberg, *Asymptotic experimental analysis for the Held-Karp traveling salesman bound*, Proceedings of the 7$^{\text{th}}$ Annual ACM-SIAM Symposium on Discrete Algorithms, 341–350, 1996.
[23] I. Krimi, R. Todosijević, R. Benmansour, M. Ratli, A. A. El Cadi, A. Aloullal, *Modelling and solving the multi-quays berth allocation and crane assignment problem with availability constraints*, J. Glob. Optim. **79** (2020), 349–373.
[24] I. Krimi, R. Benmansour, R. Todosijević, N. Mladenović, M. Ratli, *A no-delay single machine scheduling problem to minimize total weighted early and late work*, Optim. Lett. **17** (2023), 2113–2131.
[25] S. Lin, B. W. Kernighan, *An effective heuristic algorithm for the travelling salesman problem*, Oper. Res. **21** (1973), 498–516.
[26] D. S. McGeoch, L. A. Johnson, *The travelling salesman problem: a case study in local optimization*, In: E. Aarts, J. K. Lenstra (eds.), *Local Search in Combinatorial Optimization*, 215–310, Wiley, Chichester, 1997.
[27] I. Méndez-Díaz, P. Zabala, A. Lucena, *A new formulation for the traveling deliveryman problem*, Discret. Appl. Math. **156** (2008), 3223—3237.
[28] M. Mikić, R. Todosijević, D. Urošević, *Less is more: General variable neighborhood search for the capacitated modular hub location problem*, Comput. Oper. Res. **110** (2019), 101–115.
[29] A. Mjirda, R. Todosijević, S. Hanafi, P. Hansen, N. Mladenović, *Sequential variable neighborhood descent variants: an empirical study on the traveling salesman problem*, Int. Trans. Oper. Res. **24** (2017), 615–633.
[30] N. Mladenović, P. Hansen, *Variable neighborhood search*, Comput. Oper. Res. **14** (1997), 1097–1100.
[31] N. Mladenović, D. Urošević, S. Hanafi, A. Ilić, *A general variable neighborhood search for the one-commodity pickup-and-delivery travelling salesman problem*, Eur. J. Oper. Res. **220** (2012), 270–285.
[32] N. Mladenović, R. Todosijević, D. Urošević, *An efficient GVNS for solving traveling salesman problem with time windows*, Electron. Notes Discrete Math. **39** (2012), 83–90.
[33] N. Mladenović, R. Todosijević, D. Urošević, M. Ratli, *Solving the Capacitated Dispersion Problem with variable neighborhood search approaches: From basic to skewed VNS*, Comput. Oper. Res. **139** (2022), 105622.
[34] M. Ratli, D. Urošević, A. A. El Cadi, J. Brimberg, N. Mladenović, R. Todosijević, *An efficient heuristic for a hub location routing problem*, Optim. Lett. **16** (2022), 281–300.

[35] A. Salehipour, K. Sörensen, P. Goos P, O. Brysy, *Efficient GRASP+VND and GRASP+VNS metaheuristics for the traveling repairman problem*, 4OR **9**(2) 2011, 189-–209.

[36] R. Todosijević, S. Hanafi, D. Urošević, B. Jarboui, B. Gendron, *A general variable neighborhood search for the swap-body vehicle routing problem*, Comput. Oper. Res. **78** (2017), 468–479.

[37] R. Todosijević, R. Benmansour, S. Hanafi, N. Mladenović, A. Artiba, *Nested general variable neighborhood search for the periodic maintenance problem*, Eur. J. Oper. Res. **252** (2016), 385–396.

[38] R. Todosijević, R. Benmansour, M. Mladenović, S. Hanafi, N. Mladenović, I. Crévits, *Adaptive general variable neighborhood search heuristics for solving the unit commitment problem*, International Journal of Electrical Power and Energy Systems **78** (2016), 873–883.

[39] B. Y. Wu, Z. Huang, F. Zhan, *Exact algorithms for the minimum latency problem*, Inf. Process. Lett. **92**(6) (2004), 303–309.

[40] F. Zhao, S. Li, J. Sun, D. Mei, *Genetic algorithm for the one-commodity pickup-and-delivery travelling salesman problem*, Computers and Industrial Engineering **56** (2009), 1642–1648.