Discrete Applied Mathematics 346 (2024) 95–114

Contents lists available at ScienceDirect

Discrete Applied Mathematics

journal homepage: www.elsevier.com/locate/dam

A general variable neighborhood search for the traveling salesman problem with time windows under various objectives

Mengdie Ye *, Enrico Bartolini, Michael Schneider

Deutsche Post Chair–Optimization of Distribution Networks, RWTH Aachen University, Germany

ARTICLE INFO

Article history:
Received 8 February 2023
Received in revised form 22 August 2023
Accepted 5 December 2023
Available online 15 December 2023

Keywords:
Traveling salesman problem
Time windows
Uncertainty
Robust optimization
Local search

ABSTRACT

The traveling salesman problem with time windows (TSPTW) has wide practical applications in transportation and scheduling operations. We study the TSPTW in the deterministic case as well as under travel time uncertainty. We consider the three classical TSPTW variants with the objectives of minimizing cost, completion time, and tour duration. In addition, a new TSPTW variant that maximizes the minimum slack of a tour, i.e., the smallest time buffer between the arrival time and the end of the time window over all nodes visited on the tour, is introduced. We further address a variant of the TSPTW in which the arc travel times are uncertain. This problem is modeled by means of an uncertainty set, and the goal is to determine a tour that remains feasible in the worst case within a budget stipulating the sum of all travel time deviations from their nominal values. To solve all targeted problem variants, we develop a two-phase general variable neighborhood search (GVNS) that applies an efficient move evaluation approach within the local search. Extensive numerical experiments on benchmark instances from the literature show that our GVNS finds high-quality solutions that are competitive with those obtained by the state-of-the-art heuristics for all problem variants.

1. Introduction

The traveling salesman problem with time windows (TSPTW) is a very well-known *NP*-hard combinatorial optimization problem. It aims to find an optimal tour with respect to a certain objective function starting and ending at the depot and visiting a set of nodes within their given time windows exactly once. The objectives of the TSPTW depend on the goals of the specific application. The TSPTW is common in transportation applications such as school bus transportation and post deliveries, and it can also be viewed as a subproblem of the vehicle routing problem with time windows (VRPTW); in this case, the objective is typically to minimize the total cost (or travel time). The TSPTW also arises in scheduling applications: It is equivalent to model a single-machine scheduling problem, where the task is to schedule jobs on a single machine given the release and due time of each job and sequence-dependent setup times. The task of scheduling the jobs on every single machine in the job shop scheduling problem with sequence-dependent setup times, deadlines, and precedence constraints is also equivalent to solving a TSPTW [9]. In this context, the so-called makespan, i.e., the maximum completion time of all jobs, is usually the objective to be minimized. Apart from the above two well-studied

* Corresponding author.
E-mail addresses: ye@dpo.rwth-aachen.de (M. Ye), bartolini@dpo.rwth-aachen.de (E. Bartolini), schneider@dpo.rwth-aachen.de (M. Schneider).

https://doi.org/10.1016/j.dam.2023.12.006
0166-218X/

objectives, another variant in the routing context is to minimize the tour duration, i.e., the time difference between the arrival and departure times at the depot in case the departure time at the depot is not fixed in advance. For convenience, we refer to the TSPTW with cost minimization as TSPTW-C, to the TSPTW with makespan minimization as TSPTW-M, and to the TSPTW with tour duration minimization as TSPTW-D.

Inspired by the concept of earliness in scheduling applications, which is defined as the difference between the completion time of a job and its due time, we introduce a new concept called *slack* for the TSPTW. The slack of a node is defined as the time buffer between the arrival time at the node and the end of its time window. Its value indicates how far we can postpone the arrival time at the node without violating its time window. The slack of a tour is thus defined as the minimum slack over all nodes on the tour, and it indicates how far we can postpone the arrival times at all nodes without rendering the tour infeasible. The idea behind this objective is that a tour with a slack that is as large as possible can to some extent be robust against possible travel time fluctuations. Therefore, we also study this TSPTW variant maximizing the slack of the tour and we denote it as TSPTW-S.

Most papers on the TSPTW address deterministic models, in which the parameters defining the problem, such as the travel times, are assumed to be known a-priori. However, under uncertain travel times, the solutions obtained by deterministic models are likely to be infeasible when applied in practice. In this paper, we also study a TSPTW variant under travel time uncertainty and model it as a robust optimization problem. We consider the knapsack-constrained uncertainty set $\mathcal{T}_K$ that has been first introduced by Bartolini et al. [11]. $\mathcal{T}_K$ comprises all possible travel time realizations of interest, each one has a total amount of travel time deviations not larger than a predefined delay budget Δ. The resulting problem, denoted as RTSPTW($\mathcal{T}_K$), is to find a tour with minimum cost that remains feasible with respect to all travel time realizations described by $\mathcal{T}_K$, or in other words, that is feasible for up to Δ units of cumulated delay.

To the best of our knowledge, no solution method has addressed more than two TSPTW variants at the same time (e.g., [27] and [28] both study the TSPTW-C and the TSPTW-M), and it is not clear whether existing heuristics achieve good performance across multiple problem variants. Contrary to this, the method presented in this paper is designed to target all four deterministic TSPTW and the RTSPTW variants described above.

To solve the variants, we develop a two-phase general variable neighborhood search (GVNS) heuristic. The heuristic is based on the variable neighborhood search (VNS) paradigm (see, e.g., [36]) and consists of a constructive and an improvement phase. In the first phase, we build a feasible solution using a VNS heuristic. In the second phase, we try to improve this solution using a GVNS heuristic, i.e., a VNS using a variable neighborhood descent (VND) [23] as local search procedure. A preliminary version of the heuristic, which is designed only for the RTSPTW($\mathcal{T}_K$), was used by the exact algorithm of Bartolini et al. [11] to obtain valid upper bounds for the problem. Our heuristic is inspired by the two-phase GVNS proposed by Da Silva and Urrutia [17], which was originally developed to solve the TSPTW-C. Contrary to the original method, we replace the level-based random *OR-opt-1* move with a level-based destroy and repair procedure of Karabulut and Tasgetiren [28] in the perturbation phase, and we use additional neighborhoods in the VND to be able to handle multiple problem variants. Moreover, we adapt the move evaluation approach from [47] to efficiently evaluate the feasibility and profitability of a local search move, and we extend it to the concept of robust feasibility required when solving the RTSPTW($\mathcal{T}_K$).

We report the results of extensive computational experiments conducted on different benchmark sets from the literature for each of the considered variants. We compare our results with the solutions obtained by the state-of-the-art heuristics and the best-known solutions (BKS) from the literature. The results show that our two-phase GVNS is able to achieve high-quality solutions on the benchmark sets for all variants considered. In particular, our heuristic finds 13 new BKS for the TSPTW-D and one new BKS for the RTSPTW($\mathcal{T}_K$). Apart from the extensive comparison to existing heuristics, we also present a study of the influence of different algorithmic components on the performance of our algorithm. In particular, we examine the impact of each neighborhood implemented in the VND and the effect of the modified perturbation procedure.

The paper is structured as follows. We survey the related literature in Section 2. In Section 3, we provide a technical description of the TSPTW variants with different objective functions. The two-phase GVNS heuristic for solving all variants is described in Section 4. We describe the computational experiments in Section 5. Finally, Section 6 presents our conclusion.

2. Literature review

The TSPTW-C, the TSPTW-M, and the TSPTW-D have been extensively studied, and we review exact and heuristic solution methods in Section 2.1. In Section 2.2, we review papers that address the RTSPTW with a budgeted uncertainty set.

2.1. Papers on deterministic TSPTWs

Among the three deterministic TSPTW variants, the TSPTW-C is the most studied one. A variety of exact solution methods can be found in the literature. These methods are mainly based on dynamic programming (DP, see, e.g., [8,10,19]), branch-and-cut algorithms [4,13,18], and constraint programming (CP, see, e.g., [21,40]). The TSPTW-M has also been well studied. Many exact solution methods, from branch-and-bound algorithms [5,16,30] to mixed integer programming (MIP)

formulations [26,27], have been developed. The TSPTW-D has not received as much attention as the other two variants. There are two recent papers solving the TSPTW-D in an exact fashion. One is from [46], they propose a new DP algorithm that generalizes the approach presented by Baldacci et al. [10] and provide for the first time the BKS for four benchmark sets. The other one is from [32], they extend the algorithm of Tilk and Irnich [46] to deal with time-dependent costs and provide 31 new BKS for the problem. For an extensive review of the exact solution methods for deterministic TSPTW variants, we refer to Baldacci et al. [10] and Pralet [42].

Numerous heuristics have been developed for solving the TSPTW-C. Carlton and Barnes [15] solve the TSPTW-C using a tabu search (TS) heuristic that allows infeasible solutions that are handled by a static penalty function. Gendreau et al. [22] propose a construction heuristic to gradually build a feasible solution and improve it through the removal and reinsertion of all nodes. Ohlmann and Thomas [39] use compressed annealing, which is a variant of simulated annealing that relaxes the time window constraints by integrating a penalty function using a variable penalty multiplier within a stochastic search procedure. López-Ibáñez and Blum [33] propose a hybrid method called Beam-ACO that combines ant colony optimization with a tree search method called beam search. Their method is able to improve the best solutions found by Ohlmann and Thomas [39] on several benchmark sets. Beam-ACO is also adapted to the TSPTW-M by the same authors [35] and it provides the BKS for the TSPTW-M on several benchmark sets.

Several papers opt for the metaheuristic GVNS embedded into a two-phase heuristic framework to solve the TSPTW-C and the TSPTW-M. Such a two-phase heuristic is usually composed of a constructive phase, in which a feasible solution is built using a VNS, and an improvement phase, in which a GVNS is called to improve the solution returned by the previous phase. Da Silva and Urrutia [17] are the first to apply a two-phase GVNS to the TSPTW-C. Their results show the great potential of the GVNS by providing better solutions for several benchmark sets compared to those reported in [33]. Mladenović et al. [37] extend this approach by using more neighborhoods in the VND in the improvement phase. They also introduce a vector that is used to effectively verify the feasibility of a move. They are able to improve the BKS for some instances, provide better average solution quality, and reduce the average runtimes on all tested benchmarks compared to Da Silva and Urrutia [17]. Recently, Amghar et al. [2] have adapted this approach to the TSPTW-M. They apply a different order of exploring the neighborhoods in the VND than [37] to handle the makespan objective more effectively. Their computational results show that the GVNS is able to generate at least as good results as the Beam-ACO for the tested benchmarks, and provides new BKS for several instances.

An alternative approach that is competitive with the two-phase GVNS heuristic of Mladenović et al. [37] is the variable iterated greedy (VIG) algorithm of Karabulut and Tasgetiren [28] for the TSPTW-C. The algorithm of Karabulut and Tasgetiren [28] is able to provide new BKS for several benchmark instances compared to Da Silva and Urrutia [17]. For the TSPTW-M, Pralet [42] has recently developed an algorithm called iterated maximum large neighborhood search (ImaxLNS) which dominates the two-phase GVNS of Amghar et al. [2]. ImaxLNS is a combination of iterated local search (ILS) and large neighborhood search (LNS). It uses a parameter called insertion-width to control the maximum destroy size such that the repair method can remain applicable with a limited complexity. The algorithm is able to return the BKS for all tested benchmark sets in less than 1 s in the worst case for the TSPTW-M and can be applied to the time-dependent version of the TSPTW.

Heuristic approaches for the TSPTW-D are rather limited. Savelsbergh [44] is the first to address tour duration minimization in the routing context. He introduces the concept of forward time slack to help compute the tour duration and solves the problem by a local search heuristic based on edge exchanges. Favarreto et al. [20] call the problem temporal TSPTW. They develop an ant colony approach to solve the problem and report solutions for benchmark sets originally proposed for the TSPTW-C. Tilk and Irnich [46] develop a VND heuristic for the problem to generate upper bounds that can be used in their DP approach. The VND starts from a known best tour to the TSPTW-C or TSPTW-M and calls the Balas–Simonetti neighborhoods (see Section 4.2) in the local search iteratively. To the best of our knowledge, the proposed VND is the state-of-the-art heuristic for the TSPTW-D.

2.2. Papers on robust routing problems

For robust routing problems considering uncertain parameters, we mainly review the solution methods based on the budget of uncertainty model of Bertsimas and Sim [12], in which the uncertainty set encodes a budget of uncertainty defined by a cardinality constraint specifying the number of uncertain parameters that are allowed to deviate from their nominal value.

Based on this model, the robust VRPTW under travel time and (or) demand uncertainty have been extensively addressed. Exact solution methods, such as branch-and-cut algorithms [1,38] and DP approaches [31] have been proposed to solve small instances to optimality. Several heuristic approaches have also been developed. Braaten et al. [14] consider a robust VRPTW in maritime transportation in which travel times are uncertain. They use a similar uncertainty set as in [1] and propose an efficient heuristic based on adaptive LNS to solve the resulting problem. Hu et al. [24] address the VRPTW under both demand and travel time uncertainty and design a two-stage algorithm based on a modified adaptive VNS heuristic. We refer to Bartolini et al. [11] and Zhang et al. [48] for a more detailed overview of robust routing problems.

All the above robust models consider uncertainty sets that are parameterized by means of a budget value Γ that is cardinality-based and that models the maximum number of travel times whose value can simultaneously deviate from the nominal one. Contrary to this, Bartolini et al. [11] study the robust TSPTW under travel time uncertainty and introduce a

knapsack-constrained uncertainty set in which the budget value is denoted by Δ and represents the maximum cumulative delay that can be incurred on a tour. They solve the resulting problem, denoted as RTSPTW($\mathcal{T}_K$), using an exact algorithm based on column generation and DP with a state–space relaxation adapted from [10]. In the paper at hand, we study the same problem as in [11], i.e., the RTSPTW($\mathcal{T}_K$), and aim for an effective heuristic for solving it.

3. Problem description

The TSPTW can be formulated on a complete directed graph $G = (N, A)$ with $N = \{0, 1, 2, ..n\}$ as the node set and A as the arc set. Each arc $(i, j) \in A$ is associated with a cost $c_{ij} > 0$ and a travel time $t_{ij} > 0$ (it is assumed in most applications that $c_{ij} = t_{ij}$ if not specified). The travel time t_{ij} generally includes the service time at node i. Each node $i \in N$ is associated with a time window $TW_i = [e_i, l_i]$, $e_i < l_i$, which represents a time interval within which node i must be visited. Node 0 represents the depot, and we assume that $TW_0 = [0, T]$, where $T \geq l_i, \forall i = 1, \ldots, n$.

We call a path that starts from the depot 0, visits each node $i \in N\backslash\{0\}$ exactly once, and finally returns to the depot 0, a tour, and refer to it as $P = (i_0 = 0, i_1, \ldots, i_n, i_{n+1} = 0)$. As commonly done in most formulations of the problem, we allow waiting times at the nodes, which means that if the arrival time at a node i is before e_i, waiting occurs until the start of its time window. Therefore, given a tour P, the departure time from a node i_h of P with $h = 0, \ldots, n+1$, is calculated as $\tau_h^D = \max\{\tau_h^A, e_{i_h}\}$, where $\tau_h^A = \tau_{h-1}^D + t_{i_{h-1}\, i_h}$ is the arrival time at node i_h. It is convenient to assume that $\tau_0^A = \tau_0^D$ and $\tau_{n+1}^A = \tau_{n+1}^D$. A tour P is feasible if it allows to visit each node i_h of P within its time window, i.e., $e_{i_h} \leq \tau_h^A \leq l_{i_h}$, $\forall h = 0, \ldots, n+1$. The problem is to find a feasible tour with respect to a specific objective function as discussed in the following.

3.1. The TSPTW with classical objectives

Let $\mathcal{P}$ be the set of all possible tours $P = (i_0, i_1, \ldots, i_{n+1})$ in G, the TSPTW-C can be formulated as follows (see e.g., [28,33]):

$$\min_{P\in\mathcal{P}} \quad f_c(P) = \sum_{h=0}^{n} c_{i_h\, i_{h+1}}$$

$$s.t. \quad \Omega(P) = \sum_{h=0}^{n+1} \omega(i_h) = 0, \ \forall P \in \mathcal{P}$$

where $\omega(i_h) = 1$ if $\tau_h^A > l_{i_h}$, and 0 otherwise. In the above definition, the total number of violated time window constraints of a tour $P \in \mathcal{P}$, denoted by $\Omega(P)$, must be zero for feasible solutions. $f_c(P)$ is the objective function minimizing the total cost of arcs traversed along tour P.

The TSPTW-M and the TSPTW-D only differ from the TSPTW-C with respect to the objective function. Given a tour P, the objective functions of minimizing its makespan or tour duration can be respectively written as:

$$\min_{P\in\mathcal{P}} f_m(P) = \tau_{n+1}^A,$$

or

$$\min_{P\in\mathcal{P}} f_d(P) = \tau_{n+1}^A - \tau_0^D.$$

Note that in the TSPTW-D, the departure time τ_0^D at the depot is not fixed and thus is a decision variable that has to be determined for a given tour.

3.2. The TSPTW with slack maximization

In this section, we provide a formal definition of the slack of a tour. Given a path $P = (i_0, i_1, \ldots, i_p)$ starting from node $i_0 = 0$ and ending at a node $i_p \in N\backslash\{0\}$, we define the slack s_h of the hth node i_h ($0 \leq h \leq p$) of P, as the gap between the end of the time window l_{i_h} and its arrival time τ_h^A, i.e.,

$$s_h = l_{i_h} - \tau_h^A.$$

The value of s_h indicates how far the arrival time at node i_h can be shifted forward in time without causing a time window violation at i_h. We then define the slack of P, denoted as $\underline{s}(P)$, as the minimum slack over all nodes of P. More precisely, we define:

$$\underline{s}(P) = \min_{h=0,\ldots,p} s_h.$$

It is not hard to see that P can remain feasible if the arrival time at each node i_h with $0 \leq h \leq p$ is postponed by an amount that is at most $\underline{s}(P)$ units.

Consider the occurrence of a delay that will cause a forward shifting of the arrival time at each node of a tour. If the amount of delay is smaller than the value of the slack, the tour can remain feasible. Hence, we believe that the slack of a tour can in fact to a certain degree reflect the robustness of a tour under travel time uncertainty. Therefore, we consider a new TSPTW variant in which the objective function is to maximize the slack of a tour. The resulting problem, referred to as the TSPTW-S, can be formulated analogously to the TSPTW-C with the following objective function:

$$\max_{P\in\mathcal{P}} f_s(P) = \max_{P\in\mathcal{P}} \underline{s}(P) = \max_{P\in\mathcal{P}} \min_{h=0,\dots,p} s_h.$$

3.3. The robust TSPTW with a knapsack-constrained uncertainty set

In the RTSPTW($\mathcal{T}_K$), the travel time t_{ij} of each arc (i,j) is no longer deterministic but, instead, can take any value within an interval $T_{ij} = [t^0_{ij}, t^0_{ij} + \delta_{ij}]$, where t^0_{ij} is called the nominal travel time and δ_{ij} is called the maximum delay of arc (i,j). In this problem, we assume that the total amount of delay incurred by a tour P, i.e., the quantity $\sum_{(i,j)\in P} t_{ij} - t^0_{ij}$, cannot exceed an upper bound called delay budget Δ. The corresponding knapsack-constrained uncertainty set $\mathcal{T}_K$ is thus defined as follows:

$$\mathcal{T}_K = \left\{ \mathbf{t} \in \mathbb{R}^{|A|} : t_{ij} = t^0_{ij} + \xi_{ij}\delta_{ij},\ 0 \le \xi_{ij} \le 1,\ \forall (i,j) \in A,\ \sum_{(i,j)\in A} \delta_{ij}\xi_{ij} \le \Delta \right\}.$$

The goal of the RTSPTW($\mathcal{T}_K$) is to find a least-cost tour that remains feasible with respect to all travel time vectors belonging to $\mathcal{T}_K$.

Consider a path $P = (i_0, i_1, \dots, i_p)$ with $i_0 = 0$ and $i_p \in N\backslash\{0\}$ and a travel time vector $\mathbf{t} \in \mathcal{T}_K$. We define $\tau^{\mathbf{t}}_h$ as the departure time from a node i_h of P with respect to $\mathbf{t}$. It is calculated as:

$$\tau^{\mathbf{t}}_h = \max\{e_{i_h}, \tau^{\mathbf{t}}_{h-1} + t_{i_{h-1}i_h}\}, \quad \forall h = 1, \dots, p.$$

Note that the departure time $\tau^{\mathbf{t}^0}_h(P)$ with respect to the travel time vector $\mathbf{t}^0$ is equivalent to the nominal departure time $\tau^D_h(P)$ in the deterministic case. Clearly, a path P is robust feasible with respect to $\mathcal{T}_K$ if

$$\max_{\mathbf{t}\in\mathcal{T}_K} \tau^{\mathbf{t}}_h \le l_{i_h}, \quad \forall h = 1, \dots, p. \tag{1}$$

Bartolini et al. [11] provide an explicit characterization for the robust feasibility of a tour with respect to the uncertainty set $\mathcal{T}_K$, and they develop an efficient algorithm to verify the robust feasibility condition (1). Based on this, we are able to evaluate the robust feasibility of a RTSPTW($\mathcal{T}_K$) tour in the local search procedure of the proposed heuristic. A simplified version of the algorithm is available in Appendix A, and we refer to Bartolini et al. [11] for a more detailed discussion.

4. A two-phase GVNS for the TSPTW under various objectives

We propose a two-phase heuristic for solving all the considered problem variants. The heuristic is inspired by the GVNS heuristic presented in [17]. It features a different perturbation procedure and involves additional neighborhoods in the VND. We use the efficient move evaluation approach from [47] in the local search procedure, which has the advantage to track multiple values that are related to different objectives and to speed up the search process. In particular, we introduce an extension of it that can be applied to the RTSPTW($\mathcal{T}_K$).

As commonly done, the algorithm starts with a preprocessing step, in which incompatible arcs that cannot belong to any feasible solution are identified. Specifically, an arc (i,j) is incompatible if $e_i + t_{ij} > l_j$. After the preprocessing, the algorithm executes two phases. Algorithm 1 provides a pseudocode overview of our two-phase GVNS heuristic. The first is a constructive phase, which tries to find a feasible solution using a VNS (line 1). This phase iteratively calls a perturbation and a local search procedure until a feasible solution is found or the time limit is reached (see detailed procedure in Section 4.1). The second is an improvement phase, which aims to improve the feasible solution obtained by the first phase using a GVNS (lines 3 to 10), i.e., a variant of VNS with a VND as the local search (see Section 4.2). In each iteration of the GVNS, we first perform a perturbation (identical to that in the constructive phase) to the best solution found so far (line 6) and then perform a local search (line 7). The incumbent solution and the perturbation strength k are updated at the end of each iteration (line 8). The GVNS iteratively performs the above procedures until a given maximum perturbation strength k_{max} (see Section 4.1.1 for explanation) or a time limit is reached.

In the remainder of this section, we describe the details of the constructive and improvement phases in Sections 4.1 and 4.2, respectively. We explain the efficient move evaluation approach and its extension to the robust problem in Section 4.3, and we discuss several speedup techniques in Section 4.4.

Algorithm 1: Two-phase GVNS

```
1  X ← ConstructionVNS() ;                                   // Constructive phase
2  X* ← X;
3  repeat                                                    // Improvement phase
4  |  k ← 1;
5  |  repeat
6  |  |  X' ← Perturbation(X, k);
7  |  |  X'' ← VND(X');
8  |  |  NeighborhoodChange(X, X'', k);
9  |  until k > k_max;
10 until time limit is reached;
```

Algorithm 2: ConstructionVNS

```
1  X^0 ← RandomSolution();
2  repeat
3  |  X ← OrOpt1LS(X^0);
4  |  k ← 1;
5  |  repeat
6  |  |  X' ← Perturbation(X, k);
7  |  |  X'' ← OrOpt1LS(X');
8  |  |  NeighborhoodChange(X, X'', k);
9  |  until k > k_max;
10 |  UpdatePenalty(X, α);
11 until X is feasible or time limit is reached;
```

4.1. Constructive phase

To build a feasible solution, we follow the VNS-based procedure proposed in [17] but apply some modifications.

The objective function in this phase used by Da Silva and Urrutia [17] is to minimize the sum of the time window violations, i.e., the sum of all positive differences between the arrival time at each node and the end of its time window. In their algorithm, finding a feasible solution is equivalent to finding a solution (i.e., a TSPTW tour) that has an objective function value of zero. To also possibly guide the search process towards high-quality solutions (i.e., a solution with an objective value as small as possible in a minimization problem), we instead consider the original objective function of each problem variant and permit time window violations but penalize them. To this end, we use a generalized objective function which is defined as follows:

$$f_{gen}(X) = f(X) + \alpha \cdot P_{tw}(X),$$

where $f(X)$ is the objective value of a solution X depending on the problem variant, $P_{tw}(X)$ is the total time window violations of X, and α is the corresponding penalty weight. Thus, our algorithm in this phase aims at finding a solution that is both feasible ($P_{tw}(\cdot) = 0$) and of good quality.

Algorithm 2 shows the constructive phase. The VNS iterates until it finds a feasible solution or a time limit is reached. In the first step of the VNS, the algorithm generates a random solution X^0 which is possibly infeasible (line 1). Then the algorithm iterates through lines 2 to 11 to find a feasible solution using the following steps. First, a local search procedure using the *OR-opt-1* neighborhood (a neighboring solution is obtained by relocating a node, see, e.g., [37] or [2]) is applied to X^0 (line 3). The variable k, used to control the strength of the perturbation, is set to 1 (line 4). The algorithm then iteratively improves the incumbent solution X until the maximum perturbation strength k_{max} is reached (lines 5 to 9). More precisely, in each iteration, X is first destroyed and rebuilt by calling the perturbation procedure controlled by the current perturbation strength value k (line 6), and then improved by a local search using the *OR-opt-1* neighborhood (line 7). At the end of each iteration, the incumbent solution and the variable k are updated (line 8). If no feasible solution can be found for some consecutive iterations, the penalty weight α is updated (line 10). If no feasible solution can be found within the time limit in this phase, the algorithm terminates.

4.1.1. Perturbation procedure

In the perturbation, instead of using a level-based random *OR-opt-1* move as in [17], we use a level-based destroy and repair procedure similar to that of the VIG algorithm presented in [28].

This procedure consists of two parts. The first part destroys a solution by removing d nodes from it, and the second part rebuilds a new complete solution with a constructive insertion heuristic by inserting each removed node into the partial

solution. As in the perturbation procedure of Karabulut and Tasgetiren [28], we use an integer parameter $k = 1, \ldots, k_{max}$ to control the strength of a specific perturbation, where k_{max} is the maximum perturbation strength allowed. In our implementation, the destroy size d of one perturbation procedure is controlled by the corresponding perturbation strength k. To be more specific, we use a parameter μ to represent the proportion of the nodes that will be affected at the lowest perturbation strength $k = 1$. The maximum perturbation strength k_{max} can then be calculated as $k_{max} = (n-1) \times \mu$, and for each value of k, the destroy size is then determined as $d = k \times \lfloor (1/\mu) \rfloor$. For each of the removed d nodes, we consider its insertion to the position which yields the smallest increase in the objective value.

4.1.2. Local search procedure

In the local search procedure of the VNS, we consider only the *OR-opt-1* neighborhood, which consists in relocating one node to another position. Our local search implementation is of first-improvement type. This means that the incumbent solution is updated as soon as an improvement in the current neighborhood is found, and a change of the neighborhood happens only if there is no possible improvement in the current one.

To evaluate as few neighboring solutions as possible, we first investigate the moves that are more likely to yield a feasible solution. To this end, we partition the nodes of an incumbent solution into two subsets: the set of violated nodes, i.e., the nodes visited after the end of their time windows, and the set of non-violated nodes, i.e., the nodes visited before the end of their time windows. An *OR-opt-1* move can also be divided into two types: forward or backward move of a node. As a result, we can derive four types of the *OR-opt-1* move: backward move of a violated node, forward move of a violated node, backward move of a non-violated node, and forward move of a non-violated node. Da Silva and Urrutia [17] present a discussion that shows that exploring the *OR-opt-1* neighborhood in the following order leads to finding a feasible solution more efficiently than exploring the complete neighborhood without partitioning: backward move of a violated node, forward move of a non-violated node, backward move of a non-violated node backward, and forward move of a violated node. We follow this guideline and use the same order of the *OR-opt-1* moves in our implementation.

4.2. Improvement phase

If a feasible solution X can be found after the constructive phase, the heuristic tries to improve it using a GVNS (see lines 2 to 10 in Algorithm 1). The perturbation calls the same destroy and repair procedure as in the constructive phase, and the local search procedure of the GVNS uses a VND, which is a local search framework that systematically explores different neighborhoods to find a local minimum. VND often incorporates more than three neighborhoods and searches the neighborhoods in a predetermined order (see, e.g., [23]). In our implementation, the VND considers more neighborhoods than the one in [17] to address multiple problem variants more effectively.

In our VND procedure, we consider six neighborhoods for all problem variants in the following order: *OR-opt-1 backward*, *OR-opt-1 forward*, *1-opt*, *OR-opt-2 backward*, *OR-opt-2 forward*, *2-opt*. Similar to the *OR-opt-1* neighborhood, the *OR-opt-2* neighborhood is obtained by relocating a chain of two consecutive nodes (see, e.g., [37]). The *2-opt* neighborhood is formed by removing two edges and reconnecting the tour in a way that inverts the sequence of the nodes between the selected edges (see, e.g., [17]). All our neighborhoods are searched in best-improvement fashion, i.e., all possible moves in a neighborhood are evaluated and the best move that provides the greatest improvement to the solution is chosen. We have tested several other orders, especially the orders proposed in [37] and in [2], but the order shown above gives the best results for all considered problem variants.

In addition to these neighborhoods, we include the Balas–Simonetti neighborhood in our VND to improve its performance in finding high-quality solutions. Balas [7] introduced a family of large-scale neighborhoods for the TSP, denoted as $\mathcal{N}_{BS}^k$ for $k > 2$. Each of these neighborhoods, defined by a specific k, contains an exponential number of neighboring solutions, yet they can be explored efficiently. Their use in the TSPTW context was first discussed in [8]. To be self-contained, we briefly summarize the Balas–Simonetti neighborhood of the TSP.

Let $X = (x_0, x_1, \ldots, x_{n+1})$ be a TSP tour. For a parameter $k \geq 2$, $\mathcal{N}_{BS}^k(X)$ consists of all solutions $X' = (x_{\pi(0)}, x_{\pi(1)}, \ldots, x_{\pi(n+1)})$, where π is a permutation of $\{0, 1, \ldots, n+1\}$ that satisfies: (i) $\pi(0) = 0$, $\pi(n+1) = n+1$; (ii) $\pi(i) \leq \pi(j)$ for any pair of two indices $i, j \in \{1, \ldots, n\}$ such that $i + k \leq j$. This means that if a node x_i precedes a node x_j by at least k positions in solution X, x_i must still precede x_j in the neighboring solution X'. A best neighboring solution $X' \in \mathcal{N}_{BS}^k(X)$ can be found by solving a shortest path problem with the help of an auxiliary graph G_k^* in $\mathcal{O}(nk^2 2^k)$. Fig. 1 shows an example of an auxiliary graph for $k = 3$ given a solution $X = (x_0, \ldots, x_{n+1})$ with $n = 4$. In the auxiliary graph G_k^*, each position on the tour represents a stage, each stage has states (vertices) that are associated with different values α, and the states of consecutive stages are connected by arcs. Each state of a stage corresponds to a restricted permutation of the nodes around position i. In each stage, the constraints on these permutations are characterized by the values of α, which takes integer values strictly between $-k$ and k. Then, a state associated with α at stage i represents that the node $x_{i+\alpha}$ is moved from position $i + \alpha$ to position i in the neighboring solution X', i.e., $x'_i = x_{i+\alpha}$.

Every path in the graph starting from state o and ending at state d corresponds to a neighboring solution X'. In the example, the path associated with the neighboring solution $X' = (x_0, x_3, x_1, x_4, x_2, x_5)$ is depicted in bold. A detailed discussion of the general structure of the auxiliary graph can, e.g., be found in [8] or [46].

For a given k, the computational effort required to explore $\mathcal{N}_{BS}^k$ is linear in n, yet it increases rapidly for larger values of k. Therefore, we investigate two versions of the GVNS heuristic to solve all deterministic TSPTW variants: One puts more emphasis on the solution quality by including $\mathcal{N}_{BS}^k$ with a value of $k = 6$ as the last neighborhood in the VND, and we denote it as GVNS-Q; the other one considers only the six classical neighborhoods in the VND, is denoted as GVNS-S, and aims to find good solutions with a smaller amount of runtime.

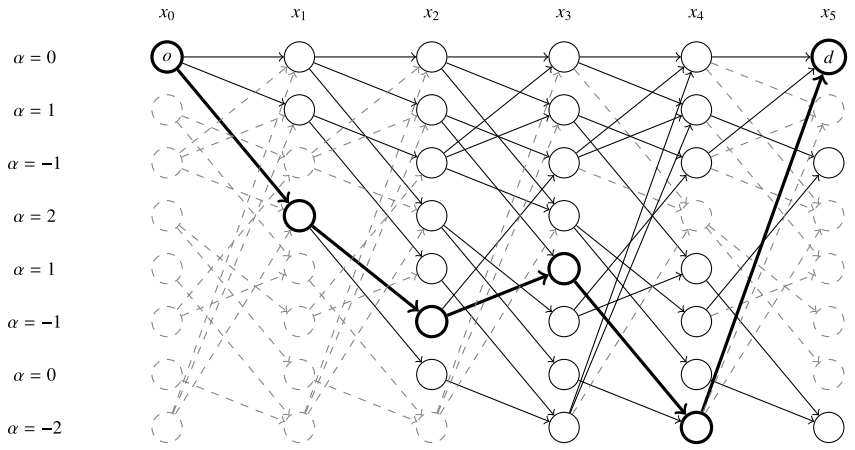

Fig. 1. Auxiliary Graph G_k^* for $k = 3$ for $\mathcal{N}_{BS}^k(X)$ of $X = (x_0, \ldots, x_5)$.
Source: Tilk and Irnich [46].

4.3. Efficient move evaluation approach

In the local search procedure, every move needs to be evaluated with respect to feasibility and profitability (a move is profitable if it improves the current solution). Checking the profitability of a move for the TSPTW-C or the RTSPTW($\mathcal{T}_K$) is easy, one only needs to compare the sum of the cost of the deleted arcs and that of the inserted arcs. For the other three problem variants, checking the profitability of a move is of the same difficulty as checking the feasibility: we need to update the arrival times at all the nodes of the tour that are involved in the move, which is time-consuming.

To speed up the evaluation of solutions in the neighborhood search, we adapt the move evaluation approach proposed by Vidal et al. [47] to our setting. It is a sequence-based approach that allows to efficiently evaluate the solutions with respect to their duration and time window violations. We first describe the approach adapted to the deterministic TSPTW variants and then discuss the modifications that are needed in the robust setting.

4.3.1. Move evaluation in deterministic TSPTWs

The approach of Vidal et al. [47] exploits the fact that all moves based on a constant number of arc exchanges or sequence relocations can be viewed as a separation of a tour into subsequences of visits, which are concatenated into a new tour (see the formal property in, e.g., [25,29,47]). For instance, the new tour obtained after applying an *OR-opt-1* move by relocating a node i to a forward position of j can be described using the concatenation of four subsequences of visits: $(0, \ldots, i-1) \oplus (i+1, \ldots, j) \oplus (i) \oplus (j+1, \ldots, n+1)$, where $\oplus$ represents the concatenate operation. Each operation is evaluated using the stored values for the involved subsequences.

The effectiveness of the approach depends substantially on the concatenation operation. Vidal et al. [47] show that in the VRPTW, many objective functions and constraints can be evaluated in amortized constant time for inter-route and intra-route moves which corresponds to moves in the TSPTW. Therefore, we can directly apply their concatenations of the distance, duration, and time window violations to our deterministic TSPTW variants. However, we still need to account for the objective value of the makespan and the slack. In the remainder, we first briefly present the definitions of the concatenations of distance *Dist*, duration *Dur*, time window violations *TW*, and corresponding earliest and latest departure times from the first node *E* and *L* in [47]. We then describe the concatenations of the makespan *M* and the slack *S* that are derived from the previous definitions.

Let $\sigma = (i, \ldots, j)$ and $\sigma' = (i', \ldots, j')$ be two subsequences of visits. The following operations are presented by Vidal et al. [47]:

$$\textit{Distance} : Dist(\sigma \oplus \sigma') = Dist(\sigma) + d_{ji'} + Dist(\sigma'), \tag{2}$$

$$\textit{Duration} : Dur(\sigma \oplus \sigma') = Dur(\sigma) + Dur(\sigma') + t_{ji'} + \Delta_{WT}, \tag{3}$$

$$TW\ \textit{violation} : TW(\sigma \oplus \sigma') = TW(\sigma) + TW(\sigma') + \Delta_{TW}, \tag{4}$$

$$\textit{Earliest} : E(\sigma \oplus \sigma') = \max\{E(\sigma') - \Delta_D, E(\sigma)\} - \Delta_{WT}, \tag{5}$$

$$\textit{Latest} : L(\sigma \oplus \sigma') = \min\{L(\sigma') - \Delta_D, L(\sigma)\} - \Delta_{TW}, \tag{6}$$

where $\Delta_D = Dur(\sigma) + t_{ji'} - TW(\sigma)$ is an auxiliary variable for calculating the time needed to reach the first node of σ', $\Delta_{WT} = \max\{e_{i'} - \Delta_D - L(\sigma), 0\}$ is the additional waiting time, and $\Delta_{TW} = \max\{E(\sigma) + \Delta_D - l_{i'}, 0\}$ is the additional time window violation after reaching the first node of σ'.

Note that in the routing context, the makespan represents the completion time of ending the tour, which can be seen as a special case of the tour duration when the starting time at the original depot is fixed. Therefore, we can define the concatenation of the makespan M using the duration Dur and the earliest departure E that ensures this minimum duration, i.e.:

$$\textit{Makespan}: \ M(\sigma \oplus \sigma') = Dur(\sigma \oplus \sigma') + E(\sigma \oplus \sigma'). \tag{7}$$

As for the slack, we recall that it indicates how far we can postpone the arrival time at each node such that the tour remains feasible. Hence, its value for the concatenated sequence can be easily calculated using the earliest and latest departure E and L, and the time window violation TW, i.e.:

$$\textit{Slack}: \ S(\sigma \oplus \sigma') = L(\sigma \oplus \sigma') - E(\sigma \oplus \sigma') - TW(\sigma \oplus \sigma'). \tag{8}$$

Based on operations (2)–(8), the evaluation of a move is to first calculate the values for the concatenations on relevant subsequences and then to evaluate feasibility and profitability with respect to different objective functions of the new sequence after the move. Hence, if the values for the concatenations on each relevant subsequence can be computed in a preprocessing step, we can perform any move evaluation in constant time.

4.3.2. Move evaluation in the RTSPTW($\mathcal{T}_K$)

In the RTSPTW($\mathcal{T}_K$), the determination of cost can easily be done using the concatenation of distance as mentioned above. To check the robust feasibility of a given tour $P = (i_0, \ldots, i_p)$, we need to calculate for each node i_h on P its worst-case departure time, which can be computed by considering the scenarios in which P starts to accumulate as much delay as possible from all possible nodes i_k with $0 \leq k < p$ (see Appendix A). This means that we need to keep track of the amount of delay that is cumulated from any possible i_k, which is non-trivial when concatenating two subsequences.

However, if we concatenate one arbitrary subsequence and another one that contains only one single node, i.e., we extend the first sequence by inserting a new node at the end of it, the values for the above concatenations in the deterministic setting can be modified in such a way that the position k of the starting node of delay on the subsequence and the cumulated delay starting from node i_k can also be stored.

Let $\sigma = (i_0, \ldots, i_h)$ be the first subsequence and $\sigma' = (i_v)$ be the second one consisting of one node i_v. We next introduce the minimum duration $Dur_k(\sigma)$, minimum time window violation $TW_k(\sigma)$, and the earliest and latest departure times $E_k(\sigma)$ and $L_k(\sigma)$ when the delay starts to accumulate from the k-th position ($0 \leq k < h$) on the subpath represented by σ. Note that these values for σ' are computed as: $Dur_k(\sigma') = 0$, $TW_k(\sigma') = 0$, $E_k(\sigma') = e_{i_v}$, and $L_k(\sigma') = l_{i_v}$. The following operations can be used to evaluate the time window violations of the new sequence:

$$Dur_k(\sigma \oplus \sigma') = Dur_k(\sigma) + Dur_k(\sigma') + t^0_{i_h i_v} + \min\{\delta_{i_h i_v}, r(\sigma, k, h)\} + \Delta^k_{WT}, \tag{9}$$

$$TW_k(\sigma \oplus \sigma') = TW_k(\sigma) + TW_k(\sigma') + \Delta^k_{TW}, \tag{10}$$

$$E_k(\sigma \oplus \sigma') = \max\{E_k(\sigma') - \Delta^k_D, E_k(\sigma)\} - \Delta^k_{WT}, \tag{11}$$

$$L_k(\sigma \oplus \sigma') = \min\{L_k(\sigma') - \Delta^k_D, L_k(\sigma)\} + \Delta^k_{TW}, \tag{12}$$

where $\delta_{i_h i_v}$ is the maximum delay on arc (i_h, i_v), $r(\sigma, k, h)$ is the residual delay that σ can possibly incur from i_h onward under the control of the delay budget Δ if the delay starts to accumulate as much as possible from i_k (see detailed explanation in Appendix A). In fact, the most important change with respect to the calculations in the deterministic case is that we use the actual travel time $t^0_{i_h i_v} + \min\{\delta_{i_h i_v}, r(\sigma, k, h)\}$ on arc (i_h, i_v) when delay starts from i_k to replace the nominal travel time that we use in the deterministic setting. The calculation of the additional waiting Δ^k_{WT} and time window violations Δ^k_{TW} after reaching the new node, as well as the auxiliary variable Δ^k_D can also be extended easily as follows: $\Delta^k_D = Dur_k(\sigma) + t^0_{i_h i_v} + \min\{\delta_{i_h i_v}, r(\sigma, k, h)\} - TW_k(\sigma)$, $\Delta^k_{WT} = \max\{E_k(\sigma') - \Delta^k_D - L_k(\sigma), 0\}$, and $\Delta^k_{TW} = \max\{E_k(\sigma) + \Delta^k_D - L_k(\sigma'), 0\}$.

Based on these operations, the robust feasibility of any move considered in this paper can be evaluated efficiently provided all concatenation values associated with the corresponding subsequences can be retrieved in constant time for all possible positions k of the starting node of delay.

4.4. Speedup techniques

As mentioned in Section 4.2, we only allow feasible solutions during the search in the improvement phase. This allows us to use specific techniques to avoid evaluating unpromising neighboring solutions.

Because the evaluation of cost can be done in $\mathcal{O}(1)$, we can discard those neighboring solutions with a larger cost in the TSPTW-C before evaluating their feasibility. This also applies to the RTSPTW($\mathcal{T}_K$). Additionally, as discussed in the move evaluation approach in Section 4.3.1, the feasibility of a TSPTW tour can be checked in constant time. Hence, in the RTSPTW($\mathcal{T}_K$), before evaluating the robust feasibility of a solution, we can filter out those that are already infeasible with respect to the deterministic case, i.e., those with positive time window violations.

In the TSPTW-M and TSPTW-D, because the objective functions are related to the arrival times at the nodes, we cannot separate the evaluation of profitability and feasibility. However, we observe that given a tour $P = (i_0, i_1, i_2, \ldots, i_n, i_{n+1})$,

if there exists positive waiting time at a node i_h, i.e., the arrival time $\tau_h^A < e_{i_h}$, relocating it to a backward position on the tour would never decrease its departure time, i.e., $\tau_h^D = e_{i_h}$, and thus would not be profitable. Hence, a neighboring solution in the TSPTW-M or TSPTW-D that is produced by moving a node with positive waiting time to a previous position on the tour can be discarded.

5. Computational experiments

We introduce the benchmark sets in Section 5.1, and we describe the computational environment and the parameter setting of our algorithm in Section 5.2. We carry out numerical experiments to (i) investigate the impact of the efficient move evaluation approach discussed in Section 4.3 (see Section 5.3), (ii) obtain a better understanding of the influence of different algorithmic components on the performance of our algorithms (see Section 5.4), and (iii) compare GVNS-S and GVNS-Q to the state-of-the-art heuristics from the literature for each variant (see Section 5.5).

5.1. Benchmark instances

For the deterministic variants, we consider in total seven benchmark sets to (i) test the performance of our algorithm on diverse instances, and (ii) provide a comprehensive set of solutions that can be compared with in future studies. The seven benchmark sets are the following:

- The *Dumas* set proposed by Dumas et al. [19] contains 135 instances with a number of nodes ranging from 20 to 200. The instances are grouped into 27 classes, each of which has five instances with the same number of nodes and the same maximum time window width.
- The *GDE* set consists of 130 instances which were obtained from the *Dumas* instances with 20 to 100 nodes by extending the time window width. This set was introduced by Gendreau et al. [22].
- The *OT* set, generated by Ohlmann and Thomas [39], contains 25 instances that were obtained from the *Dumas* instances with 150 and 200 nodes in the same way as the *GDE* instances.
- The *DaSilva* set was proposed by Da Silva and Urrutia [17] and consists of 125 instances with a number of nodes ranging from 200 to 400. The instances are grouped into 25 classes in the same way as the *Dumas* instances.
- The *AFG* set contains 50 asymmetric instances with 10 to 233 nodes introduced by Ascheuer [3].
- The *Pesant* set was proposed by Pesant et al. [40] and consists of 27 instances with 19 to 44 nodes. These instances were derived from Solomon's RC2 VRPTW instances [45].
- The *Potvin* set introduced by Potvin and Bengio [41] contains 30 instances that were also derived from Solomon's RC2 instances. The number of nodes per instance ranges from 3 to 45.

All benchmarks are available at http://lopez-ibanez.eu/tsptw-instances.

For the RTSPTW($\mathcal{T}_K$), the only available benchmark set has been proposed by Bartolini et al. [11]. These instances are derived from the *GDE* instances with 20 to 80 nodes. They are divided into two groups, called *GDE-D* and *GDE-I*, which differ with respect to the definition of the maximum arc travel time. For each instance in the former group, the maximum arc travel time is proportional to its nominal value, while for each instance in the latter group, it equals the nominal value plus a random independent number. In each group, there are three different uncertainty levels that are represented by three different budget values $\Delta = 20$, 40, and 60. The resulting set contains a total of 630 instances and is available at https://pubsonline.informs.org/doi/abs/10.1287/trsc.2020.1011.

5.2. Computational environment and parameter setting

Both GVNS-S and GVNS-Q are implemented in C++ and compiled with GCC release 9.1. All experiments were conducted on a single core of a computing cluster with an Intel(R) Xeon(R) E5-2430v2 processor at 2.50 GHz with 64 GB RAM under CentOS 7. For each instance, GVNS-S and GVNS-Q were run 15 times, and each run was executed sequentially in a single thread.

5.2.1. Parameter tuning

We use the iterated F-race algorithm [6], as implemented in the `irace` software package [34], to tune the parameters of our algorithms. The tuning process incorporates the four deterministic TSPTW variants and all seven benchmark sets. Each benchmark set is partitioned into two sets: a training set, comprising 20% of the instances, is used for tuning, and a testing set, consisting of the remaining 80%, is used for testing the tuned parameter configurations. As mentioned in Section 4.2, GVNS-S and GVNS-Q are fundamentally similar in design, with the key difference being that GVNS-Q includes the Balas–Simonetti neighborhood in the VND. Because the other neighborhoods in the VND in GVNS-Q follow the same underlying structure as those in GVNS-S, we decided to tune the parameters using only GVNS-S, and then also use these parameters for GVNS-Q. Using the same parameter configuration for both variants also guarantees that any difference in performance between them can be attributed solely to the impact of the additional Balas–Simonetti neighborhood. Table 1 provides an overview of the parameters and their domains considered for tuning. In all experiments, we restrict

Table 1
Parameters and their tuning domain.

Parameter	Definition	Domain
α_0	Initial value of the penalty weight α	{5, 10, 15, 20}
δ	Value to be multiplied with the penalty weight α	{1.1, 1.2, . . . , 2.0}
η	# of consecutive iterations without changing feasibility status of the best solution found	{1, 2, . . . , 7}
μ	Proportion of the nodes affected in the perturbation phase	{0.1, 0.2, 0.25, 0.3}

Table 2
Results of the parameter configurations of GVNS-S.

Configuration	Parameter setting				Friedman test	
	α_0	δ	η	μ	p-value (5% significance level)	Δ_R
1	10	2.0	2	0.25		11
2	15	1.6	5	0.25		9
3	5	1.3	5	0.25	0.133	26.5
4	5	1.2	2	0.20		68.5
5	10	1.2	5	0.25		0

the maximum value of α to 1000 during the search. The time limit for finding a feasible solution in the constructive phase is set to 30 s, and the total time limit is set to 60 s.

Every single run of the `irace` yields a configuration for the aforementioned parameters being tuned. We set the stopping criterion of a single tuner run to 1000 individual runs of GVNS-S. To find a generic configuration for all considered variants, we use a generalized objective function, taking into account the four objectives (cost, makespan, duration, and slack) and the runtime T, to evaluate the "cost" value of a tuner run:

$$cost = a \cdot f_{agg} + T, \tag{13}$$

where $f_{agg} = (f_c + f_m + f_d - f_s)/4$ denotes the average value of the four objectives (we subtract the value of f_s because TSPTW-S is a maximization problem), and a is a constant used to guarantee a lower cost value for a solution that either has a smaller average value f_{agg} or requires less runtime T given the same average value f_{agg}. A value of a that satisfies the above requirements should be greater than any possible value of the runtime (see, e.g., [35]). Because the time limit of each GVNS-S run is 60 s, we set $a = 100$. We repeated the `irace` tuner five times and thus obtained five parameter configurations of GVNS-S as shown in Table 2.

5.2.2. Evaluation of the tuned configuration

We then evaluate five versions of GVNS-S defined by the found parameter configurations on the instances in the testing set. As in the tuning process, for each configuration, we calculate one single value using Eq. (13) for each evaluation run, and a single evaluation run consists of 1000 runs of GVNS-S.

To provide statistical information on the results, we apply the non-parametric Friedman test to the values returned by these evaluation runs. In our case, the Friedman test ranks the various configurations for each instance and detects differences across the testing set. Table 2 reports the resulting p-value of the Friedman test with a significance level of 5%. Given that the value is greater than 0.05, there is no significant difference between the performance of the configurations. To select the preferred configuration, we then look at the total rank (i.e., the sum of ranks) for each configuration across all instances (the lower the rank the better). We report in column Δ_R in Table 2 the difference between the total rank of each configuration and the lowest total rank. From the results, we see that configuration 5 has the lowest rank, followed by configurations 2 and 1. As a result, we choose configuration 5 as the final parameter setting for both GVNS-S and GVNS-Q.

5.3. Impact of the efficient move evaluation approach

To investigate the effect of the adapted move evaluation approach, we compare the runtimes of its efficient implementation (see Section 4.3) with a straightforward implementation of the move evaluation for various problem variants. The straightforward move evaluation tentatively performs a move and tests its feasibility and profitability by recalculating the related information, e.g., arrival time at each node, of the solution obtained after the move.

The comparison is conducted on the *Dumas*, *GDE* and *OT* instances for the deterministic TSPTW variants and on the *GDE-D* and *GDE-I* instances with a medium uncertainty level for the RTSPTW($\mathcal{T}_K$). Fig. 2 presents the results in terms of runtimes of the two implementations. The runtimes are given in seconds as the average over 15 runs for all instances with the same number of nodes. Because for different objective functions, the average runtimes of the respective implementation share the same increasing trend for an increasing number of nodes, the runtimes for the deterministic variants were further aggregated over all considered objective functions. The comparison in Fig. 2 demonstrates the efficiency of the adapted move evaluation approach, and as expected, its advantage clearly increases with the number of nodes in the instances.

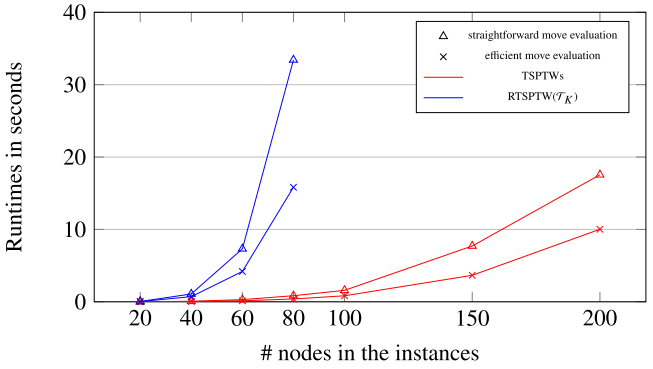

Fig. 2. Runtimes (in seconds) for the two implementations.

5.4. Analysis of algorithmic components

To gain deeper insights into the behavior of our algorithms, we carry out a series of experiments to evaluate the influence of the algorithmic components included in our GVNS heuristics. In the experiments, we use two testing sets: one with 100 representative instances from all seven benchmark sets for the deterministic TSPTW variants, and another with 100 instances from the *GDE-D* and *GDE-I* sets with a medium uncertainty level for the RTSPTW($\mathcal{T}_K$).

First, we study the influence of the Balas–Simonetti neighborhood. For this purpose, we conducted experiments with GVNS-S and GVNS-Q. Second, we analyze the importance of each classical neighborhood implemented in the VND (see Section 4.2). To this end, we performed experiments with six versions of GVNS-S, each of which excludes one specific neighborhood in the VND. In the presentation of results, these versions are denoted, respectively, as no *OR-opt-1 (b)* (i.e., neighborhood *OR-opt-1 backward* is switched off, analogous for other versions), no *OR-opt-1 (f)*, no *1-opt*, no *OR-opt-2 (b)*, no *OR-opt-2 (f)*, and no *2-opt*. Finally, we examine the effect of the level-based destroy and repair procedure in the perturbation phase. Specifically, we assess the performance difference when implementing the perturbation phase as outlined in [17,37]. This involves executing experiments with a version of GVNS-S in which the perturbation phase applies level-based random *OR-opt-1* moves. The corresponding algorithmic version is denoted as random-move perturbation. These experiments with all algorithmic versions were conducted for all deterministic TSPTW variants and the RTSPTW($\mathcal{T}_K$), and 15 runs for each algorithmic version were executed.

To clearly show whether the algorithmic components of our GVNS heuristics have significant impact on the performance in terms of both solution quality and runtime, we compare GVNS-S and each of the aforementioned algorithmic versions pairwise using the Wilcoxon signed-rank test with a significance level of 5%. For each instance in the testing set, the metric used for measuring the solution quality is the percentage gap of the average solution found over all 15 runs to the best-known solution (BKS) (computed as $100 \cdot \frac{avg-BKS}{BKS}\%$, where avg denotes the average solution values), and the time metric is the average runtime of the 15 runs. In our tests, the null hypothesis for each pairwise test is that GVNS-S performs statistically identical to the other algorithmic version in the comparison. We can conclude that an algorithmic component (such as the Balas–Simonetti neighborhood) significantly impacts the performance if evidence allows us to reject the null hypothesis at the given significance level, i.e., p-value < 0.05. Additionally, to show which algorithmic version is superior, we calculate the average values of the average gaps and the runtimes over all instances for each version. Table 3 summaries the p-values obtained from the Wilcoxon signed-rank tests and the average values of the gaps and runtimes for all pairs of comparisons under consideration.

The table is vertically structured into two main blocks, the upper one reporting the results of the three classical deterministic TSPTW variants, i.e., TSPTW-C, TSPTW-M, and TSPTW-D, and the lower one representing the TSPTW-S and the RTSPTW($\mathcal{T}_K$). For each problem variant, the results concerning the gap metric are reported in column *gap* and those concerning the time metric are reported in column t. To be more specific, we show in column Δ_{gap} the difference of the average gaps between the two algorithmic versions in the comparison (calculated as the gap of the first algorithmic version subtracting that of the second one). A "−" ("+") in this column indicates a negative (positive) difference and thus suggests that the first algorithmic version provides better (worse) average solutions. The difference in average runtimes is provided in an analogous way in column Δ_t, and a "−" ("+") in this column indicates that the first algorithmic version requires shorter (longer) runtime. For both gap and time metrics, we also show in column $p < 0.05$ the results of the pairwise Wilcoxon signed-rank test. A "✓" in this column indicates that there is a significant difference between the performance of the two algorithmic versions.

The results highlight a significant influence of the Balas–Simonetti neighborhood on the solution quality of our algorithm across all problem variants. Specifically, GVNS-Q is able to consistently find superior solutions for all variants (as indicated by the "−" in column Δ_{gap}) compared to GVNS-S. The effect of classical neighborhoods on our algorithm's performance varies with different problem variants. For instance, the *1-opt* neighborhood has no substantial influence

Table 3
Results of the pairwise comparison between algorithmic versions.

Classical TSPTW variants												
Algorithmic pairs	TSPTW-C				TSPTW-M				TSPTW-D			
	gap		*t*		*gap*		*t*		*gap*		*t*	
	Δ_{gap}	$p < 0.05$	Δ_t	$p < 0.05$	Δ_{gap}	$p < 0.05$	Δ_t	$p < 0.05$	Δ_{gap}	$p < 0.05$	Δ_t	$p < 0.05$
GVNS-Q vs. GVNS-S	−	✓	+	✓	−		+	✓	−	✓	+	✓
GVNS-S vs. no OR-opt-1 (b)	−	✓	−		−	✓	+		−		−	✓
GVNS-S vs. no OR-opt-1 (f)	−	✓	+	✓	−	✓	−		−		−	
GVNS-S vs. no 1-opt	−		+	✓	−		+		−	✓	+	
GVNS-S vs. no OR-opt-2 (b)	−		−		−		−		−		−	
GVNS-S vs. no OR-opt-2 (f)	−	✓	+		−		−		−		−	
GVNS-S vs. no 2-opt	−	✓	+	✓	−	✓	−		−	✓	+	
GVNS-S vs. random-move perturbation	−	✓	+	✓	−		+	✓	−	✓	+	✓

Newly proposed TSPTW variants					Robust TSPTW			
Algorithmic pairs	TSPTW-S				Algorithmic pairs	RTSPTW($\mathcal{T}_K$)		
	gap		*t*			*gap*		*t*
	Δ_{gap}	$p < 0.05$	Δ_t	$p < 0.05$		Δ_{gap}	$p < 0.05$	Δ_t	$p < 0.05$
GVNS-Q vs. GVNS-S	−	✓	+	✓					
GVNS-S vs. no OR-opt-1 (b)	−		+	✓	GVNS-S vs. no OR-opt-1 (b)	−	✓	+	
GVNS-S vs. no OR-opt-1 (f)	−		+	✓	GVNS-S vs. no OR-opt-1 (f)	−		+	✓
GVNS-S vs. no 1-opt	−		+	✓	GVNS-S vs. no 1-opt	−	✓	+	✓
GVNS-S vs. no OR-opt-2 (b)	−		+	✓	GVNS-S vs. no OR-opt-2 (b)	−	✓	+	✓
GVNS-S vs. no OR-opt-2 (f)	−		+	✓	GVNS-S vs. no OR-opt-2 (f)	−		+	✓
GVNS-S vs. no 2-opt	−	✓	+	✓	GVNS-S vs. no 2-opt	−	✓	+	
GVNS-S vs. random-move perturbation	−	✓	+	✓	GVNS-S vs. random-move perturbation	−	✓	−	✓

on the TSPTW-C, TSPTW-M, and TSPTW-S, but it does significantly affect the solution quality of the TSPTW-D and RTSPTW($\mathcal{T}_K$). The results in column Δ_{gap} reveal that excluding any of the classical neighborhoods tends to diminish solution quality in comparison to GVNS-S. Furthermore, the level-based destroy and repair procedure in the perturbation also significantly influences the solution quality of GVNS-S across all problem variants, consistently leading to substantial improvements.

In terms of runtime, the influence of the Balas–Simonetti neighborhood on the speed of our algorithm is significant, enabling it within GVNS-Q always lead to longer runtime when compared to GVNS-S. This aligns with our expectation that GVNS-S is designed for speed while GVNS-Q emphasizes quality. Additionally, the runtime is notably influenced by the level-based destroy and repair procedure. GVNS-S frequently takes longer than the version with a random-move perturbation. However, these extended runtimes are justified by the substantial enhancement in solution quality. When looking at the classical neighborhoods, their impact on the speed of our algorithm differs with various problem variants, and no general trend can be derived from these results regarding whether disabling one neighborhood consistently leads to reduced runtimes. However, most variations in runtime are usually small enough to be considered negligible.

In summary, the analysis reveals that each investigated algorithmic component, including all the neighborhoods and the specific perturbation procedure, plays a vital role in the performance of our algorithms. By designing two versions of the GVNS heuristic, one focus on speed (GVNS-S) and the other on quality (GVNS-Q), we are able to tailor our methods to different needs.

5.5. Performance comparison on the different TSPTW variants

In this section, we compare the performance of GVNS-S and GVNS-Q to the state-of-the-art heuristics from the literature on the benchmark sets described in Section 5.1. Sections 5.5.1–5.5.5 provide summary tables of the comparisons for the four deterministic TSPTW variants and the RTSPTW($\mathcal{T}_K$), respectively. Each row of these tables presents the averaged statistics of the runs performed for each set over all its instances. In our experiments, GVNS-S and GVNS-Q were run 15 times for each instance, respectively. The name of the set is given in the first column. The second column represents the BKS, which corresponds to the best-known value from the literature (not necessarily optimal) or the best value from any run of our algorithms if it is better or if the respective instance has not be considered so far. The third column reports the number of instances with proven optimal BKS from the literature, out of the total number of instances in the corresponding set. The remaining columns are divided into two main blocks, the first reports the performance of the start-of-the-art heuristics, and the second reports the performance of GVNS-S and GVNS-Q. Because the best solutions reported by the algorithms from the literature were obtained for different numbers of runs, we base the comparison on the average solution quality and runtimes over the runs performed by the different algorithms. To ensure a fairer runtime comparison to the results from the literature, which were obtained by computers with different CPUs, we report in the tables the respective processor, its Passmark score for a single core (see www.cpubenchmark.net), and the resulting factor

Table 4
Aggregated results for all seven benchmark sets on the TSPTW-C.

Inst. set	BKS	#opt/#inst	MTU				KT				GVNS-S				GVNS-Q			
			#opt	#BKS	gap_a%	t_{avg}	#opt	#BKS	gap_a%	t_{avg}	#opt	#BKS	gap_a%	t_{avg}	#opt	#BKS	gap_a%	t_{avg}
Dumas	579.5	135/135	135	135	0.01	0.3	135	135	**0.00**	0.8	135	135	0.03	0.2	135	135	**0.00**	0.5
GDE	432.4	108/130	108	130	0.02	0.7	108	130	**0.00**	0.8	107	126	0.11	0.2	108	129	0.04	0.6
OT	737.7	1/25	1	25	**0.08**	10.9	1	25	0.20	27.5	1	13	0.26	4.8	1	19	0.10	10.8
DaSilva	12 135.1	–/25[a]	–	22	**0.07**	13.5	–	21	0.15	63.5	–	8	0.33	3.5	–	10	0.09	5.4
AFG	5 550.8	47/50	46	49	**0.00**	3.4	–	–	–	–	30	30	0.09	0.4	38	38	0.02	1.6
Pesant	623.7	26/27	26	27	**0.00**	0.2	–	–	–	–	26	27	0.02	0.0	26	27	0.01	0.1
Potvin	634.7	26/30	26	30	**0.01**	0.4	–	–	–	–	25	30	0.13	0.0	25	30	0.13	0.0
Processor type			Core i3-380M				Core i5-540M				Xeon E5-2430v2							
Processor speed			2.53 GHz				2.53 GHz				2.50 GHz							
Passmark score			1016				1156				1483							
Time factor			0.7				0.8				1.0							

[a] Best-known values for the *DaSilva* set are averaged values for each group containing five instances with the same number of nodes and same time window width.

for the runtime correction. We use this factor to translate all runtimes into a common measure, and the runtimes reported in the tables based on different CPUs are already scaled. Note that values in bold in all tables indicate the best results in the comparison, and a "–" indicates that no solution is available for the respective method on the corresponding benchmark set. For each problem variant, the detailed comparison on each individual set is available in Appendix B.

5.5.1. Results for the TSPTW-C

Our GVNS-S and GVNS-Q have been tested on all seven benchmark sets for the TSPTW-C. We compare the results obtained by our methods to those obtained by the GVNS heuristic of Mladenović et al. [37], denoted as MTU, and to those obtained by the VIG algorithm of Karabulut and Tasgetiren [28], denoted as KT. MTU was run 15 times on each instance in the sets *AFG*, *Potvin* and *Pesant*, and 30 times on each one in the remaining sets, and 25 runs of KT were performed on each instance. As mentioned before, the comparison is therefore based on the average results over the runs of each algorithm.

Table 4 summarizes the comparison. The BKS of this problem correspond to the best-known values from the literature as reported at http://lopezibanez.eu/tsptw-instances. For the considered benchmark sets, most of the instances haven been solved to optimality (see, e.g., [43]). Column *#opt/#inst* reports the information about the number of instances in which an optimal solution is known, out of the total number of instances in the corresponding set. Results of each set are presented using the following metrics: the metric *#opt* indicates the number of instances in which the best solution found by the respective method corresponds to the known optimal solution, while the metric *#BKS* represents the number of instances where the best solution is equal to the known BKS; the gap metric gap_a% reports the percentage gap of the average solution found by the respective method to the BKS (computed as $100 \cdot \frac{avg-BKS}{[BKS]}$%, where *avg* denotes the average solution found by the respective method); and the time metric t_{avg} represents the average value of the runtimes in seconds over all runs.

According to the obtained results, GVNS-Q achieves a better average solution quality and has a lower variability in most of the runs than GVNS-S, but it requires more runtimes. In terms of quality, MTU shows the best performance, being able to find good average solutions for all sets with a largest gap to the BKS within 0.08%. KT performs better than MTU on the sets *Dumas* and *GDE* but is dominated by MTU on the other two sets included in their experiments. GVNS-S performs generally worse than KT and MTU on all sets. GVNS-Q is able to find the BKS for the *Dumas* set in all runs, but for the other sets, it is dominated by MTU. However, the gaps between the average solutions found by GVNS-S and the BKS are always below 0.33% and those provided by GVNS-Q are always below 0.13%. Moreover, GVNS-Q finds better average solution values than KT on the *OT* and *DaSilva* instances. Additionally, for the sets *Dumas*, *GDE*, *Pesant*, and *Potvin*, GVNS-Q (GVNS-S) only fails to find one (two) optimal solution(s). In terms of runtimes, both our methods outperform MTU and KT, especially on the large instances in the sets *OT* and *DaSilva*. In comparison to MTU (KT), the average runtime of GVNS-S is reduced by 69% (88%), and that of GVNS-Q is reduced by 36% (79%).

Because our average solution quality is not competitive to MTU and KT on the *OT*, *DaSilva*, and *AFG* instances, we test GVNS-S and GVNS-Q on these three sets by allowing longer runtimes. Specifically, we repeat the whole GVNS procedure in the improvement phase (which includes calling the perturbation and the VND for all possible k values until k_{max}) until there has been no improvement for η_{rep} consecutive times. We compared the results obtained by our methods using increasing values of η_{rep} and decided to use $\eta_{rep} = 10$ to achieve a balanced performance regarding the solution quality and the runtimes. The results are shown in Table 5.

Both GVNS-S and GVNS-Q improve in terms of solution quality, identifying a greater number of instances in which the best solutions found match the BKS. Furthermore, both algorithms are able to obtain better average solutions with smaller gaps to the BKS. In particular, GVNS-Q is now able to obtain better average values than not only KT but also MTU for the *OT* and *DaSilva* instances. Regarding the runtimes, both GVNS-S and GVNS-Q are still dominating KT. GVNS-S is still faster than MTU on the *DaSilva* and *AFG* instances, but GVNS-Q now requires more runtime than MTU on all three sets.

Table 5
Aggregated results for the *OT*, *DaSilva* and *AFG* instances on the TSPTW-C ($\eta_{rep} = 10$).

Inst. set	BKS	#opt/#inst	MTU				KT				GVNS-S				GVNS-Q			
			#opt	#BKS	gap_a%	t_{avg}	#opt	#BKS	gap_a%	t_{avg}	#opt	#BKS	gap_a%	t_{avg}	#opt	#BKS	gap_a%	t_{avg}
OT	737.7	1/25	1	25	0.08	10.9	1	25	0.20	27.5	1	18	0.14	15.9	1	22	**0.06**	26.3
DaSilva	12 135.1	–/25[a]	–	22	0.07	13.5	–	21	0.15	63.5	–	10	0.21	9.5	–	14	**0.06**	15.9
AFG	5 550.8	47/50	46	49	**0.00**	3.4	–	–	–	–	34	34	0.05	1.5	42	42	0.01	5.3
Processor type			Core i3-380M				Core i5-540M				Xeon E5-2430v2							
Processor speed			2.53 GHz				2.53 GHz				2.50 GHz							
Passmark score			1016				1156				1483							
Time factor			0.7				0.8				1.0							

[a] Best-known values for the *DaSilva* set are averaged values for each group containing five instances with the same number of nodes and same time window width.

Table 6
Aggregated results for all seven benchmark sets on the TSPTW-M.

Inst. set	BKS	#opt/#inst	ACG				P				GVNS-S				GVNS-Q			
			#opt	#BKS	RPD_a	t_{avg}	#opt	#BKS	RPD_a	t_{avg}	#opt	#BKS	RPD_a	t_{avg}	#opt	#BKS	RPD_a	t_{avg}
Dumas	664.4	135/135	134	134	**0.00**	0.1	135	135	**0.00**	0.0	134	134	0.02	0.1	135	135	**0.00**	0.2
GDE	575.0	125/130	125	130	0.01	0.1	125	130	**0.00**	0.0	125	130	0.01	0.0	125	130	**0.00**	0.3
OT	996.2	22/25	21	24	0.02	2.8	22	25	**0.00**	0.0	22	25	**0.00**	0.8	22	25	**0.00**	2.8
AFG	8865.4	48/50	48	50	**0.00**	0.1	48	50	**0.00**	0.0	47	49	0.01	0.1	48	50	**0.00**	0.4
Pesant	794.9	27/27	27	27	0.11	0.3	27	27	**0.00**	0.1	25	25	0.15	0.0	27	27	0.06	0.1
Potvin	694.9	28/30	28	30	0.01	0.3	28	30	**0.00**	0.1	27	28	0.12	0.0	27	29	0.12	0.0
DaSilva	15 869.8*	–/125	–	–	–	–	–	–	–	–	–	125	**0.00**	0.4	–	125	**0.00**	1.2
Processor type			Core i7-3770				Xeon E5-2660v3				Xeon E5-2430v2							
Processor speed			3.40 GHz				2.60 GHz				2.50 GHz							
Passmark score			2071				1816				1483							
Time factor			0.6 (= 0.4[a] × 1.4)				1.2				-							

[a] Runtimes reported in [2] were already scaled by multiplying 2.5, we retrieve the actual runtimes by first multiplying 0.4.

5.5.2. Results for the TSPTW-M

For the TSPTW-M, GVNS-S and GVNS-Q have also been tested on all seven sets, and the results are compared to those obtained by the GVNS heuristic of Amghar et al. [2], denoted as ACG, and to those obtained by the ImaxLNS of Pralet [42], denoted as P. Note that ACG was run 15 times and P was run 5 times on each instance of the six sets except the *DaSilva* set. The comparison is again based on the average results over all corresponding runs.

The summary of the comparison is shown in Table 6. For all sets except the *DaSilva* set, the BKS correspond to the optimal values reported by Rudich et al. [43] for the instances for which the optimal solution is known (the number is again shown in column *#opt/#inst*), and to the best-known values reported in [42] for those with no optimal solutions available. For the *DaSilva* instances, the BKS correspond to the best values found across our methods. The results are presented using the metrics *#opt* and *#BKS* as described above, the gap metric RPD_a, and the time metric t_{avg}. RPD_a denotes the average value of the relative percentage deviation (RPD) over all runs, where $RPD = 100 \cdot \frac{value-BKS}{BKS}$%, and *value* is the objective value of every single run. This measure was first used by López-Ibáñez and Blum [33] to assess the average solution quality and also reported in [2,42].

In terms of quality, P provides the best average solution values for all sets included in their experiments. GVNS-Q only fails to find the BKS in all runs for the *Pesant* and *Potvin* sets with a largest average RPD of 0.12%. GVNS-S performs slightly worse than GVNS-Q with a larger average RPD value of 0.15%. ACG is on par with GVNS-Q on the sets *Dumas* and *AFG*, and is better than GVNS-Q on the *Potvin* instances. Both GVNS-S and GVNS-Q outperform ACG with respect to the average solution quality obtained over all runs for the sets *GDE* and *OT*. Additionally, GVNS-S only fails to find the optimal solutions for four instances in the *Dumas*, *AFG*, *Pesant*, and *Potvin* sets, while GVNS-Q is able to find all optimal solutions except for one *Potvin* instance. In terms of runtimes, P is the fastest method on average, but it is slightly slower than our methods on the *Pesant* and *Potvin* sets. GVNS-S is faster than ACG on all sets included in their experiments, especially on the *OT* set, reducing the runtime by approximately 71%.

We also applied our methods to solve the TSPTW-M on the *DaSilva* instances, and we provide in Table 6 the solutions for these instances (indicated by $*$) for the first time. The details of the results for these instances are available in Appendix B.

5.5.3. Results for the TSPTW-D

For the TSPTW-D, both GVNS-S and GVNS-Q were again applied to all seven sets. For the *AFG*, *GDE*, *OT*, and *Potvin* instances, the results are shown in comparison with the VND heuristic of Tilk and Irnich [46], denoted as TI. Note that TI executes one run of their VND comprising Balas–Simonetti neighborhoods with different k values. To have a fair comparison, we compare their results with our average results over 15 runs.

Table 7
Aggregated results for all seven benchmark sets on the TSPTW-D.

Inst. set	BKS	#opt/#inst	TI				GVNS-S					GVNS-Q				
			#opt	#BKS	gap_a%	t_{avg}	#opt	#BKS	#imp	gap_a%	t_{avg}	#opt	#BKS	#imp	gap_a%	t_{avg}
GDE	503.6	119/130	91	92	0.52	7.7	109	120	–	0.15	0.1	115	126	–	**0.06**	0.5
OT	946.2	12/25	10	15	0.42	42.0	9	19	8	0.04	1.2	10	22	8	**0.03**	5.7
AFG	8418.1	50/50	46	46	0.01	1.8	48	48	–	0.02	0.1	49	49	–	**0.00**	0.4
Potvin	648.36	30/30	15	15	1.21	1.3	19	19	–	0.45	0.0	22	22	–	**0.31**	0.1
Dumas	659.8*	–/135	–	–	–	–	–	134	–	0.03	0.1	–	135	–	**0.01**	0.3
DaSilva	15 825.5*	–/125	–	–	–	–	–	125	–	**0.00**	0.4	–	125	–	**0.00**	1.3
Pesant	709.9*	–/27	–	–	–	–	–	22	–	0.46	0.0	–	27	–	**0.28**	0.1
Processor type			Core i7-2600				Xeon E5-2430v2									
Processor speed			3.40 GHz				2.50 GHz									
Passmark score			1741				1483									
Time factor			1.2				1.0									

Table 8
Aggregated results for all seven benchmark sets on the TSPTW-S.

Inst. set	BKS	#opt/#inst	CP optimizer				GVNS-S					GVNS-Q				
			#feas	#opt	gap_b%	t_{avg}	#opt	#BKS	#imp	gap_b%	t_{avg}	#opt	#BKS	#imp	gap_b%	t_{avg}
Dumas	7.1	100/135	104	100	0.93	179.4	100	135	2	**0.00**	0.2	100	135	2	**0.00**	0.3
GDE	56.4	102/130	108	102	0.12	243.8	102	129	2	0.03	0.3	102	130	2	**0.00**	0.6
OT	55.3	13/25	18	13	0.20	681.5	13	25	2	**0.00**	8.3	13	25	2	**0.00**	10.8
AFG	433.8	23/50	50	23	0.08	1090.7	23	50	1	**0.00**	1.1	23	50	1	**0.00**	1.8
Pesant	59.9	14/27	25	14	2.49	942.5	14	27	3	**0.00**	0.0	14	27	3	**0.00**	0.1
Potvin	73.4	15/30	27	15	0.46	937.5	15	30	3	**0.00**	0.0	15	30	3	**0.00**	0.1
DaSilva	14.6	–/125	–	–	–	–	–	125	–	2.46	1.7	–	125	–	**0.00**	3.2

Table 7 summaries the comparison on the four sets. Note that the BKS used in the comparison are the optimal values reported in [32,46] for those instances with optimal solutions available, and the best values found across the methods in comparison for the remaining instances. Apart from the metrics used in Table 4, we provide in column *#imp* the number of instances with new BKS found by our methods, out of the total number of instances that were included in the experiments in [32]. A "—" in this column indicates that no new BKS has been found by our methods.

According to the obtained results, we observe that compared to TI, GVNS-S already significantly narrows the average gaps to the BKS, and GVNS-Q further expands this advantage. More specifically, the gaps between the average solutions found by GVNS-S and the BKS are always within 0.46%, and those provided by GVNS-Q are further improved to be always below 0.31%. Moreover, for the *OT* set, both GVNS-S and GVNS-Q provide 8 new BKS. Note that these instances could not be solved to optimality by Lera-Romero et al. [32]. In addition, GVNS-S and GVNS-Q find the best solutions that correspond to the BKS for 8 and 51 more instances than TI, respectively. Regarding the runtime, GVNS-Q is slower than GVNS-S by approximately 73%. In comparison to TI, the average runtime of GVNS-S is reduced by about 97%, and that of GVNS-Q is reduced by 88%.

Additionally, we provide new solutions for the 10 *GDE* instances that were not included in both Tilk and Irnich [46] and Lera-Romero et al. [32]. We also report for the first time the solutions for the *Dumas*, *Pesant*, and *DaSilva* instances for the TSPTW-D (indicated by $*$ in Table 7). Detailed results of these instances can be found in Appendix B.

5.5.4. Results for the TSPTW-S

For the TSPTW-S, we also report results for both GVNS-S and GVNS-Q on all seven sets. Because the TSPTW-S is investigated for the first time, no comparison methods are available. Therefore, we solved all instances of the TSPTW-S using ILOG CPLEX CP optimizer (constraint programming) with a time limit of 1800 s, and we compare the results of GVNS-S and GVNS-Q to those found by the CP optimizer.

Table 8 presents the results. We do not compare the results of GVNS-S and GVNS-Q to those of the CP optimizer on the *DaSilva* instances because the CP optimizer is only able to solve less than 20 (out of 125) instances feasibly within the given time limit. For other sets, the comparison only considers the instances in which a feasible solution has been found by the CP optimizer. This information is reported in column *#feas*. The metrics *#opt*, *#BKS*, and *#imp* report the same information as described above, respectively. For this problem, we report in column gap_b% the percentage gap of the best solution found by the method over all runs to the BKS (computed as $100 \times \frac{best-BKS}{BKS}$%, where *best* denotes the best solution found by the respective method).

We recall that the TSPTW-S is a maximization problem. To be consistent with the comparisons for other problem variants, the gap metric is computed as $gap_b\% = \frac{BKS-best}{BKS} \times 100\%$. Hence a gap greater than zero still represents a deterioration. According to the results, GVNS-Q succeeds in finding the BKS for all sets. GVNS-S only fails to provide the BKS for the *GDE* and *DaSilva* instances, however, the results are still within 2.46% to the BKS. Compared to the CP

Table 9
Aggregated results for GDE instances with up to 80 customers on the RTSPTW($\mathcal{T}_K$)-T ($\eta_{rep} = 10$).

Set GDE-D																	
	$\Delta = 20$						$\Delta = 40$						$\Delta = 60$					
		GVNS-S						GVNS-S						GVNS-S				
Inst.	#opt/#inst	#opt	#imp	gap_b%	gap_a%	t_{avg}	#opt/#inst	#opt	#imp	gap_b%	gap_a%	t_{avg}	#opt/#inst	#opt	#imp	gap_b%	gap_a%	t_{avg}
n20	25/25	25	–	0.00	0.00	0.1	25/25	25	–	0.00	0.00	0.1	25/25	25	–	0.00	0.04	0.1
n40	25/25	25	–	0.00	0.00	0.8	25/25	25	–	0.00	0.00	0.7	25/25	25	–	0.00	0.02	0.8
n60	24/25	24	–	0.00	0.01	4.1	23/25	23	–	0.01	0.03	4.1	22/25	22	–	0.00	0.04	4.9
n80	28/30	28	–	0.00	0.03	14.5	24/30	24	–	0.00	0.06	16.4	22/30	22	1	0.04	0.06	20.2
Set GDE-I																		
	$\Delta = 20$						$\Delta = 40$						$\Delta = 60$					
		GVNS-S						GVNS-S						GVNS-S				
Inst.	#opt/#inst	#opt	#imp	gap_b%	gap_a%	t_{avg}	#opt/#inst	#opt	#imp	gap_b%	gap_a%	t_{avg}	#opt/#inst	#opt	#imp	gap_b%	gap_a%	t_{avg}
n20	25/25	25	–	0.00	0.00	0.1	25/25	25	–	0.00	0.02	0.1	21/25	21	–	0.00	0.01	0.1
n40	25/25	25	–	0.00	0.00	0.8	25/25	25	–	0.00	0.01	0.7	20/25	20	–	0.00	0.05	0.8
n60	24/25	24	–	0.00	0.01	4.2	24/25	24	–	0.00	0.00	4.1	15/25	15	–	0.00	0.00	4.7
n80	25/30	25	–	0.00	0.01	15.5	24/30	24	–	0.01	0.03	15.8	23/30	23	–	0.00	0.02	20.0

optimizer, both our methods are able to find new BKS for one instance in the set *AFG*, two in the *Dumas*, *GDE*, and *OT* sets, and three in the *Pesant* and *Potvin* sets. Moreover, both GVNS-S and GVNS-Q are able to find all known optimal solutions. Regarding the runtime, both methods require only a small amount of time to find solutions with good quality in comparison with the CP optimizer.

5.5.5. Results for the RTSPTW($\mathcal{T}_K$)

We decided to use only GVNS-S to solve the RTSPTW($\mathcal{T}_K$). We refrain from applying the GVNS-Q to the RTSPTW($\mathcal{T}_K$) because first, adapting the Balas–Simonetti neighborhood to the robust case with respect to $\mathcal{T}_K$ is non-trivial, and second, we observe a non-negligible runtime increasing when solving the deterministic TSPTW variants using GVNS-Q, and last but most important, GVNS-S already achieves very good performance on the RTSPTW($\mathcal{T}_K$) in the preliminary experiments. Based on the experiments for the TSPTW-C with different η_{rep} values, we decided to apply GVNS-S with $\eta_{rep} = 10$ on both *GDE-D* and *GDE-I* instances aiming for improved solution quality.

The results obtained by GVNS-S are compared to the BKS provided by the exact algorithm of Bartolini et al. [11]. Table 9 presents a summary of the comparison. The table is divided into two blocks, one reporting the results on the set GDE-D, the other on the GDE-I. Considering the fact that some instances are proven infeasible or cannot be solved feasibly within a given time limit by the exact algorithm, we only report in each row aggregated information on the instances whose optimal solutions are known (the number is given in column *#opt/#inst*). Like in the comparisons for the deterministic TSPTW variants, the metric *#opt* represents the number of instances for which an optimal solution value has been found by GVNS-S, and *#imp* reports the number of instances with new BKS identified by GVNS-S. We also report the gap metrics gap_b% and gap_a%, and the time metric t_{avg}.

The obtained results show that GVNS-S is highly successful in solving the RTSPTW($\mathcal{T}_K$) on instances with up to 80 customers. Looking at the instances of both sets GDE-D and GDE-I with a value of Δ up to 60, GVNS-S is able to find 569 optimal solution values out of the 571 found by the exact algorithm of Bartolini et al. [11]. Moreover, GVNS-S is able to find a new BKS for one instance "n80w140.002" with $\Delta = 60$. For those instances on which GVNS-S failed to find the BKS, the average gap between the best solutions obtained by GVNS-S and the BKS is always below 0.06%, and the average runtime of GVNS-S is always below 20 s for each instance class.

6. Conclusions

We study three variants of the deterministic TSPTW in which the objective functions are travel cost minimization, makespan minimization, and tour duration minimization. We introduce a new TSPTW variant with tour slack maximization, to provide solutions that are robust against delay up to a certain degree. We also address the robust TSPTW under travel time uncertainty which is defined by the knapsack-constrained uncertainty set $\mathcal{T}_K$ of Bartolini et al. [11].

We develop a two-phase GVNS heuristic for solving all described variants. We show how to efficiently evaluate feasibility and profitability of a local search move by adapting the move evaluation approach that was originally introduced by Vidal et al. [47], and we also extend the approach to efficiently check the robust feasibility in the RTSPTW($\mathcal{T}_K$). The effectiveness of the adapted approach is proven in the numerical experiments.

We conduct a comprehensive evaluation of the influence of the individual algorithmic components on the performance of our GVNS heuristic. Our findings demonstrate that each component significantly affects both solution quality and runtimes. This underscores the validity of our decision to design two algorithmic versions that emphasize speed and quality, respectively. We compare the proposed heuristic to the state-of-the-art algorithms from the literature. We report

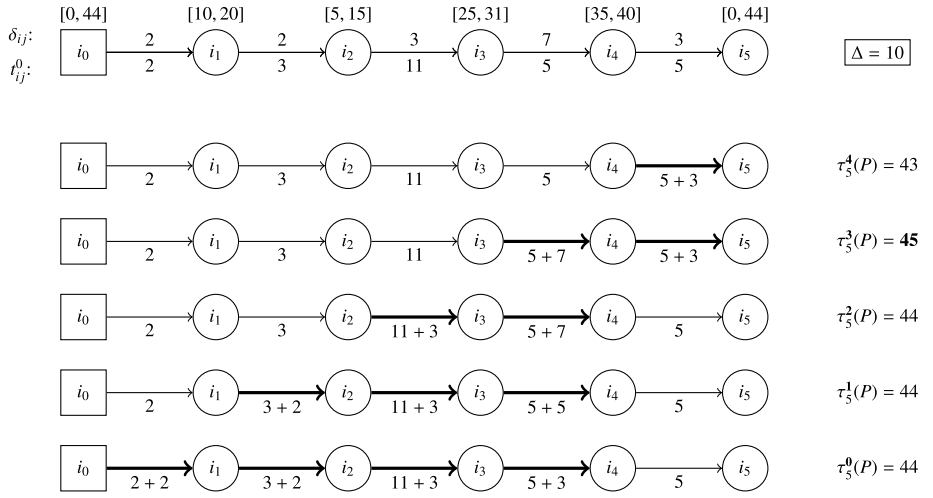

Fig. A.3. Illustration of the robust feasibility algorithm applied to a path $P = (i_0, \ldots, i_5)$. *Source:* Bartolini et al. [11].

the experimental results on different benchmark sets for all problem variants. Overall, our heuristic is successful in finding high-quality solutions within a reasonable amount of runtime for each problem variant, and is able to provide for the TSPTW-D, TSPTW-S, and the RTSPTW($\mathcal{T}_K$) new BKS for several instances.

Data availability

Data will be made available on request.

Acknowledgments

We thank the authors of Mladenović et al. [37] and those of Kara et al. [27] for providing additional information on their computational experiments that helped us to make fairer comparisons.

Appendix A. Algorithm for testing the robust feasibility with respect to $\mathcal{T}_K$

This section briefly discusses the $O(n^2)$ algorithm proposed by Bartolini et al. [11] for efficiently testing the robust feasibility of a path $P = (i_0, \ldots, i_p)$ with respect to the uncertainty set $\mathcal{T}_K$. The algorithm can be described as follows. Let τ^k_h be the latest departure time from a node i_h of P with respect to a travel time vector $\mathbf{t} \in \mathcal{T}_K$ modeling a scenario where P starts to accumulate delays from a node i_k and each arc (i, j) on the subpath $(i_k, \ldots, i_h)$ (with $0 \le k < h$) incurs its maximum allowed delay, i.e., $t_{ij} = t^0_{ij} + \min\{\delta_{ij}, \Delta - \delta(P, k, i)\}$, where $\delta(P, k, i) = \min\{\Delta, \sum_{r=k}^{i-1} \delta_{i_r i_{r+1}}\}$ denotes the maximum cumulated delay starting from node i_k until i under the control of Δ. More briefly, τ^k_h is the departure time from i_h of P when P starts to accumulate as much delay as possible from a node i_k with $0 \le k < h$. To simplify the notation, we use $r(P, k, i)$, called the maximum residual delay of P starting from node i onward, to represent the value of the term $\Delta - \delta(P, k, i)$. The entire set of departure times τ^k_h can be calculated using the following recursion:

$$\tau^k_h = \left\{ e_{i_h}, \tau^k_{h-1} + t^0_{i_{h-1} i_p} + \min\left\{ r(P, k, h-1), \delta_{i_{h-1} i_h} \right\} \right\}, \quad k = h-1, \ldots, 0, \tag{A.1}$$

with $\tau^k_k = \tau^D_k$ for all $0 \le k < h$ as the initialization. The worst-case departure time from i_h of P, denoted as $\hat{\tau}_h$, is then defined as:

$$\hat{\tau}_h = \max_{k=0,\ldots,h-1} \tau^k_h. \tag{A.2}$$

A tour P is robust feasible if and only if $\hat{\tau}_h \le l_{i_h}$ for each $1 \le h \le p$. In this case, to test the robust feasibility of a tour P with n nodes, it is enough only to consider n scenarios identified by all possible nodes from which P starts to accumulate the maximum delay.

Fig. A.3 shows an example to calculate the worst-case departure time from node i_5 of the path using the above algorithm. The delay budget Δ is 10 units. The lower part of the figure depicts five scenarios in which the path starts to accumulate as much delay as possible starting from nodes i_4 to i_0, respectively. The arcs (to each possible node i) taking the maximum allowed delay (calculated as $t^0_{ij} + r(P, k, i)$) in each scenario represented by k ($k = 0, \ldots, 4$) are marked in bold. In this example, the scenario yielding the worst-case departure time $\hat{\tau}_5$ is the one in which the path starts to accumulate as much delay as possible from node i_3.

Appendix B. Supplementary data

Supplementary material related to this article can be found online at https://doi.org/10.1016/j.dam.2023.12.006.

References

[1] A. Agra, M. Christiansen, R. Figueiredo, L.M. Hvattum, M. Poss, C. Requejo, The robust vehicle routing problem with time windows, Comput. Oper. Res. 40 (2013) 856–866.
[2] K. Amghar, J.-F. Cordeau, B. Gerdron, A General Variable Neighborhood Search Heuristic for the Traveling Salesman Problem with Time Windows Under Completion Time Minimization, Technical Report, CIRRELT-2019-29, 2019.
[3] N. Ascheuer, Hamiltonian Path Problem in the On-Line Optimization of Flexible Manufacturing System (Ph.D. thesis), Technical University Berlin, Germany, 1996.
[4] N. Ascheuer, M. Fischetti, M. Grötschel, Solving the asymmetric traveling salesman problem with time windows by branch-and-cut, Math. Program. 90 (3) (2001) 475–506.
[5] E.K. Baker, Technical note–An exact algorithm for the time-constrained traveling salesman problem, Oper. Res. 31 (5) (1983) 938–945.
[6] P. Balaprakash, M. Birattari, T. Stützle, Improvement strategies for the F-Race algorithm: Sampling design and iterative refinement, in: Hybrid Metaheuristics: 4th International Workshop, HM 2007, Dortmund, Germany, October 8-9, 2007. Proceedings 4, Springer, 2007, pp. 108–122.
[7] E. Balas, New classes of efficiently solvable generalized traveling salesman problems, Ann. Oper. Res. 86 (1999) 529–558.
[8] E. Balas, N. Simonetti, Linear time dynamic-programming algorithms for new classes of restricted TSPs: A computational study, INFORMS J. Comput. 13 (1) (2001) 56–75.
[9] E. Balas, N. Simonetti, A. Vazacopoulos, Job shop scheduling with setup times, deadlines and precedence constraints, J. Sched. 11 (4) (2008) 253–262.
[10] R. Baldacci, A. Mingozzi, R. Roberti, New state-space relaxations for solving the traveling salesman problem with time windows, INFORMS J. Comput. 24 (3) (2012) 356–371.
[11] E. Bartolini, D. Goeke, M. Schneider, M. Ye, The robust TSPTW under knapsack-constrained travel time uncertainty, Transp. Sci. 55 (2) (2021) 371–394.
[12] D. Bertsimas, M. Sim, The price of robustness, Oper. Res. 52 (2004) 35–53.
[13] N. Boland, M. Hewitt, D.M. Vu, M. Savelsbergh, Solving the traveling salesman problem with time windows through dynamically generated time-expanded networks, in: International Conference on AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, Springer, 2017, pp. 254–262.
[14] S. Braaten, O. Gjønnes, L.M. Hvattum, G. Tirado, Heuristics for the robust vehicle routing problem with time windows, Expert Syst. Appl. 77 (2017) 136–147.
[15] W.B. Carlton, J.W. Barnes, Solving the traveling salesman problem with time windows using tabu search, IIE Trans. 28 (8) (1996) 617–629.
[16] N. Christofides, A. Mingozzi, P. Toth, State-space relaxation procedure for the computation of bounds to routing problems, Networks 11 (2) (1981) 145–164.
[17] R.F. Da Silva, S. Urrutia, A general VNS heuristic for the traveling salesman problem with time windows, Discrete Optim. 7 (4) (2010) 203–211.
[18] S. Dash, O. Günlük, A. Lodi, A. Tramontani, A time bucket formulation for the traveling salesman problem with time windows, INFORMS J. Comput. 24 (1) (2012) 132–147.
[19] Y. Dumas, J. Desrosiers, E. Gelinas, M.M. Solomon, An optimal algorithm for the traveling salesman problem with time windows, Oper. Res. 43 (2) (1995) 367–371.
[20] D. Favarreto, E. Moretti, P. Pellegrini, An ant colony system approach for variants of the traveling salesman problem with time windows, J. Inf. Oper. Sci. 27 (1) (2006) 35–54.
[21] F. Focacci, A. Lodi, M. Milano, A hybrid exact algorithm for the TSPTW, INFORMS J. Comput. 14 (4) (2002) 403–417.
[22] M. Gendreau, A. Hertz, G. Laporte, M. Stan, A generalized insertion heuristic for the traveling salesman problem with time windows, Oper. Res. 46 (3) (1998) 330–335.
[23] P. Hansen, N. Mladenović, Variable neighborhood search: Principles and applications, European J. Oper. Res. 130 (3) (2001) 449–467.
[24] C. Hu, J. Lu, X. Liu, G. Zhang, Robust vehicle routing problem with hard time windows under demand and travel time uncertainty, Comput. Oper. Res. 94 (2018) 139–153.
[25] S. Irnich, A unified modeling and solution framework for vehicle routing and local search-based metaheuristics, INFORMS J. Comput. 20 (2) (2008) 270–287.
[26] I. Kara, T. Derya, Formulation for minimizing tour duration of the traveling salesman problem with time windows, Procedia Econ. Finance 26 (2015) 1026–1034.
[27] I. Kara, O.N. Koc, F. Altiparmak, B. Dengiz, New integer linear programming formulation for the traveling salesman problem with time windows: Minimizing tour duration with waiting times, Optimization 62 (10) (2013) 1309–1319.
[28] K. Karabulut, M.F. Tasgetiren, A variable iterated greedy algorithm for the traveling salesman problem with time windows, Inform. Sci. 279 (2014) 383–395.
[29] G.A.P. Kindervater, M.W.P. Savelsbergh, Vehicle routing: Handling edge exchanges, in: Local Search in Combinatorial Optimization, Princeton University Press, 1997, pp. 337–360.
[30] A. Langevin, M. Desrochers, J. Desrochers, S. Gélinas, F. Soumis, A two-commodity flow formulation for the traveling salesman and the makespan problem with time windows, Networks 23 (7) (1993) 631–640.
[31] C. Lee, K. Lee, S. Park, Robust vehicle routing problem with deadlines and travel time/demand uncertainty, J. Oper. Res. Soc. 63 (9) (2012) 1294–1306.
[32] G. Lera-Romero, J.J. Miranda-Bront, F.J. Soulignac, Dynamic programming for the time-dependent traveling salesman problem with time windows, INFORMS J. Comput. 34 (6) (2022) 3292–3308.
[33] M. López-Ibáñez, C. Blum, Beam-ACO for the traveling salesman problem with time windows, Comput. Oper. Res. 37 (9) (2010) 1570–1583.
[34] M. López-Ibáñez, J. Dubois-Lacoste, L.P. Cáceres, M. Birattari, T. Stützle, The irace package: Iterated racing for automatic algorithm configuration, Oper. Res. Perspect. 3 (2016) 43–58.
[35] C.B. Manuel López-Ibáñez, J.W. Ohlmann, B.W. Thomas, The traveling salesman problems with time windows: Adapting algorithms from travel-time to makespan optimization, Appl. Soft Comput. 13 (9) (2013) 3806–3815.
[36] N. Mladenović, P. Hansen, Variable neighborhood search, Comput. Oper. Res. 24 (11) (1997) 1097–1100.
[37] N. Mladenović, R. Todosijević, D. Urošević, An efficient general variable neighborhood search for large travelling salesman problem with time windows, Yugosl. J. Oper. Res. 23 (1) (2013) 19–30.
[38] P. Munari, A. Moreno, J.D.L. Vega, D. Alem, J. Gondzio, R. Morabito, The robust vehicle routing problem with time windows: Compact formulation and branch-price-and-cut method, Transp. Sci. 53 (4) (2019) 1043–1066.

[39] J.W. Ohlmann, B.W. Thomas, A compressed-annealing heuristic for the traveling salesman problem with time windows, INFORMS J. Comput. 19 (1) (2007) 80–90.
[40] G. Pesant, M. Gendreau, J.-Y. Potvin, J.-M. Rousseau, An exact constraint logic programming algorithm for the traveling salesman problem with time windows, Transp. Sci. 32 (1) (1998) 12–29.
[41] J.-Y. Potvin, S. Bengio, The vehicle routing problem with time windows – Part II: Genetic search, INFORMS J. Comput. 8 (2) (1996) 165–172.
[42] C. Pralet, Iterated maximum large neighborhood search for the traveling salesman problem with time windows and its time-dependent version, Comput. Oper. Res. 150 (2023) 106078.
[43] I. Rudich, Q. Cappart, L.-M. Rousseau, Improved Peel-and-Bound: Methods for generating dual bounds with multivalued decision diagrams, 2023, arXiv preprint arXiv:2302.05483.
[44] M.W. Savelsbergh, The vehicle routing problem with time windows: Minimizing route duration, ORSA J. Comput. 4 (2) (1992) 146–154.
[45] M.M. Solomon, Algorithms for the vehicle routing and scheduling problems with time window constraints, Oper. Res. 35 (2) (1987) 254–265.
[46] C. Tilk, S. Irnich, Dynamic programming for the minimum tour duration problem, Transp. Sci. 51 (2) (2017) 549–565.
[47] T. Vidal, T.G. Crainic, M. Gendreau, C. Prins, A hybrid genetic algorithm with adaptive diversity management for a large class of vehicle routing problems with time windows, Comput. Oper. Res. 40 (1) (2013) 475–489.
[48] Z. Zhang, Y. Zhang, R. Baldacci, Effective exact solution framework for routing optimization with time windows and travel time uncertainty, 2022, https://optimization-online.org/?p=18939.