Effective neighborhood search with optimal splitting and adaptive memory for the team orienteering problem with time windows

Youcef Amarouche, Rym Nesrine Guibadj, Elhadja Chaalal, Aziz Moukrim

Version of Record: https://www.sciencedirect.com/science/article/pii/S0305054820301568
Manuscript_5bfd122f3aa659e8f19a6724a02fd281

Effective Neighborhood Search with Optimal Splitting and Adaptive Memory for the Team Orienteering Problem with Time Windows

Youcef Amarouche[a,b,*], Rym Nesrine Guibadj[c], Elhadja Chaalal[b,1], Aziz Moukrim[b]

[a] *Agence de l'environnement et de la Maîtrise de l'Energie 20, avenue du Grésillé - BP 90406, 49004, Angers Cedex 01, France*
[b] *Sorbonne universités, Université de technologie de Compiègne, CNRS, Heudiasyc UMR 7253 CS 60319, 60203 Compiègne cedex, France*
[c] *Université du Littoral Côte d'Opale, EA 4491 -LISIC - Laboratoire d'Informatique Signal et Image de la Côte d'Opale, F-62228 Calais, France*

Abstract

The Team Orienteering Problem with Time Windows (TOPTW) is an extension of the well-known Orienteering Problem. Given a set of locations, each one associated with a profit, a service time and a time window, the objective of the TOPTW is to plan a set of routes, over a subset of locations, that maximizes the total collected profit while satisfying travel time limitations and time window constraints. Within this paper, we present an effective neighborhood search for the TOPTW based on (1) the alternation between two different search spaces, a *giant tour search space* and a *route search space*, using a powerful splitting algorithm, and (2) the use of a long term memory mechanism to keep high quality routes encountered in elite solutions. We conduct extensive computational experiments to investigate the contribution of these components, and measure the performance of our method on literature benchmarks. Our approach outperforms state-of-the-art algorithms in terms of overall solution quality and computational time. It finds the current best known solutions, or better ones, for 89% of the literature instances within reasonable runtimes. Moreover, it is able to achieve better average deviation than state-of-the-art algorithms within shorter computation times. Moreover, new improvements for 57 benchmark instances were found.

Keywords: Routing, Team Orienteering Problem, Time windows, Adaptive memory, Splitting algorithm, Heuristics

1. Introduction

In the Team Orienteering Problem (TOP), we are given a transportation network in which a starting and an ending point are specified. The network connects a set of points that correspond to customer locations. Each one of them is associated with a profit and a service time. For each pair of locations, a travel time is specified. The aim of the problem is to find a fixed number of disjoint paths from the starting point to the final destination through a subset of locations, each not exceeding a given time limit, that maximizes the

*Corresponding author
Email addresses: youcef.amarouche@hds.utc.fr (Youcef Amarouche), rym.guibadj@univ-littoral.fr (Rym Nesrine Guibadj), de_chaalal@esi.dz (Elhadja Chaalal), aziz.moukrim@hds.utc.fr (Aziz Moukrim)
[1] Present address: Orange Labs Networks, 22300 Lannion, France.

total profit collected from visiting customers. Fig. 1 shows an example of a TOP instance with two available vehicles, and its solution. Each customer is represented with a circle area proportional to its profit. Given two available vehicles, the selected customers and their service order are shown in Fig. 1b.

(a) Example of a TOP instance.

(b) Example of a TOP solution.

Figure 1: Team Orienteering Problem.

In this paper, we consider the Team Orienteering Problem with Time Windows (TOPTW), a natural extension of the TOP motivated by different practical situations. Possible applications of the problem range from logistics (Golden et al., 1987; Tang and Miller-Hooks, 2005) to leisure related applications like tourism (Vansteenwegen et al., 2009). In the TOPTW, each customer must be visited within a predefined time interval, specified by an earliest and a latest time, into which the service must start. We assume that the time windows are *hard* constraints. This means that early arrivals to a location are permitted, but the agent must wait for it to be "open" before the service can start. Late arrivals, however, are not allowed.

Herein, we propose an effective approach that follows the basic structure of a Multi-Start Iterated Local Search (MS-ILS) to solve the TOPTW. We design a rather straightforward method that is able to effectively explore the search space to achieve high-quality solutions within very short computational times.

- First, we investigate the alternation between two search spaces: a *giant tour search space* and a *route search space*. In the *route space*, solutions are represented as genuine TOPTW solutions i.e., a set of feasible routes, one for each vehicle in use, while in the *giant tour space*, they are represented with an ordered list of customers with no route delimiters. The transition from the *giant tour search space* to the *route search space* is achieved using a powerful split algorithm. This idea is a follow up on preliminary work carried out by Guibadj and Moukrim (2014). They proposed a Memetic Algorithm for the TOPTW, which uses a *giant tour representation* for encoding individuals. MS-ILS, however, uses the giant tour representation more efficiently within an algorithm that is conceptually simpler and much faster than the MA.

- Second, we integrate an adaptive memory mechanism to overcome the drawbacks due to pure multi-start heuristics being memoryless. The adaptive memory is used to store individual routes extracted

from diverse high quality solutions. These routes are then combined to construct promising new starting solutions at each iteration of the multi-start algorithm.

- Third, we conduct extensive computational experiments to investigate the contribution of these components to the search performance of a basic MS-ILS, and measure the performance of our approach on literature benchmarks. The obtained results show that our MS-ILS outperforms state-of-the-art algorithms in terms of solution quality and computation time. It is able to find the current best-known solutions, or better ones, for 78% of the available benchmark instances. It achieves an overall average relative gap of 0.30% and 0.26% on the two benchmarks of the literature, respectively, while being faster than most state-of-the-art algorithms. Additionally, it was able to improve the solutions of 57 instances for which no optimal solutions have yet been found. In comparison, the previous best performing approach in the literature finds 65% of the current best-known solutions, and achieves a relative gap of 0.80% and 0.34% respectively, on the two benchmarks.

- Finally, we show that our algorithm can be tuned to either favor solution quality at the cost of more computational effort, or to considerably reduce computation times while maintaining good solution quality. As such, it can serve as a good basis for future developments on more complex variants of selective vehicle routing problems.

The remainder of this paper is organized as follows. Section 2 provides a brief overview of the literature related to the TOPTW, and Section 3 gives a formal description of the problem. Section 4 gives a detailed description of our proposed algorithm. First, the general framework of the method is introduced, and then each of its aspects is described in detail, including the solution representation, the optimal split procedure, and other components and parameters. The effectiveness of our approach is shown in Section 5. Finally, Section 6 concludes this paper and provides possible directions for future research.

2. Literature review

Orienteering problems are well-known NP-hard optimization problems, and solving them within a reasonable amount of time may prove to be rather difficult.

2.1. Exact solution methods

There are relatively few studies that deal with exact methods for the OPTW and TOPTW, and most of them are designed for the OPTW. Righini and Salani (2009) solve the OPTW using a dynamic programming (DP) algorithm with *Decremental State Space Relaxation* (DSSR) (Righini and Salani, 2008), a relaxation of the problem so that customers can be visited more than once. The solution process consists in solving the DSSR, and then incrementally tightening the relaxed problem by disallowing multiple visits to a critical customer set of increasing size. Duque et al. (2015) extend the pulse algorithm (Lozano and Medaglia, 2013; Cabrera et al., 2020) to solve the OPTW. The algorithm is presented as a general-purpose framework for

hard shortest path problems, and includes two novel pruning strategies to discard sub-optimal and infeasible solutions. The proposed algorithm performs better than the dynamic programming approach of Righini and Salani (2009). The authors, however, only reported results on a subset of the benchmark instances.

The first exact method for the TOPTW was presented by Tae and Kim (2015). They proposed a Branch-&-Price (B&P) algorithm where the pricing problem is expressed as a Resource Constrained Elementary Shortest Path Problem (RCESPP). They tested their approach on the benchmark instances of Righini and Salani (2008), but omitted those of Montemanni and Gambardella (2009), stating their difficulty, and those of Vansteenwegen et al. (2009) since their optimal solutions are already known. More recently, Gedik et al. (2017) introduced an exact approach for the TOPTW based on a constraint programming (CP) formulation of the problem. Compared to the B&P, the CP-optimizer was able to solve benchmark instances from Montemanni and Gambardella (2009) and Vansteenwegen et al. (2009) which, were omitted by Tae and Kim (2015), and provides good upper bounds within smaller runtimes, for the instance of Righini and Salani (2008). However, the B&P is able to provide optimal solutions for more instances from the latter set.

2.2. Heuristic solution methods

Most of the research on orienteering problems with time windows focuses on the design of heuristic approaches. The fact that profits and travel times are independent, and that a good solution with respect to one criterion is often unsatisfactory with respect to the other, make it more difficult to devise consistently good heuristics to solve orienteering problems, despite them being seemingly simple (Gendreau et al., 1998). The presence of time window constraints increases this difficulty since it becomes harder to pinpoint the customers that should be included in the solution. Since the OPTW is a special case of the TOPTW where the fleet consists of a single vehicle, recent contributions tend to not make the distinction between single and multi-vehicle variants. Apart from the algorithm of Gunawan et al. (2015a), the heuristic methods that were introduced during the last decade were designed to solve both the OPTW and the TOPTW.

Montemanni and Gambardella (2009) proposed a heuristic method to solve the TOPTW using an Ant Colony System (ACS) algorithm. The ACS was later improved by Montemanni et al. (2011) who identified some of its drawbacks and provided ways to overcome them by only considering the best solution found during the construction phase, and only applying the local search procedure on solutions on which it has not yet been applied. Vansteenwegen et al. (2009) introduced a fast and simple Iterated Local Search (ILS) for the problem. Their algorithm was made fast and simple by only keeping the *insert* and *shake* steps. The authors also introduced a new data set with more difficult instances constructed from the instances of Solomon (1987) and Cordeau et al. (1997). Tricoire et al. (2010) defined the Multi-Period Orienteering Problem with Multiple Time Windows (MuPOPTW) as a means for scheduling sale representatives to visit customers. The MuPOPTW is a generalization of the TOPTW where visits to customers span a period of time (days), and where multiple time windows are allowed. As such, each customer may be visited on a different day, and may have several time windows for each given day. To solve the MuPOPTW, the authors designed a Variable Neighborhood Search (VNS) algorithm, which also provides good solutions for the TOPTW.

However, their algorithm requires relatively large computation times. An algorithm that combines a Greedy Randomized Adaptive Search Procedure (GRASP) and an Evolutionary Local Search algorithm (ELS) was introduced by Labadie et al. (2011). Their method uses simple constructive heuristics inside the GRASP to generate distinct initial solutions, which are then improved using the ELS algorithm. In another study, Labadie et al. (2012) applied the concept of granularity to a VNS approach in order to reduce the size of the local search neighborhoods to moves that are more likely to lead to good quality solutions. Using the dual optimal solutions of an LP-problem, they partition the arc set into intervals of granularity from which they choose those with the most promising arcs. Two Simulated Annealing based algorithms for the TOPTW were presented by Lin and Yu (2012). The first one is a Fast Simulated Annealing (FSA) that is directed towards applications that require quick responses, while the second one is a Slow Simulated Annealing (SSA) that focuses on solution quality. Souffriau et al. (2013) introduced the Multi-Constraint Team Orienteering Problem with Multiple Time Windows (MC-TOP-MTW), a variant of the TOPTW that includes additional knapsack constraints to limit node selection. Such constraints can be used, for example, to model entrance fees in the case of tourist trip planning applications. The authors proposed a hybridization of GRASP and ILS, and evaluated the performance of their algorithm on benchmarks for the related problems, including the TOPTW. An iterative framework comprising three components, namely I3CH, was proposed by Hu and Lim (2014). The first two components are a Local Search (LS) and a Simulated Annealing (SA) that are used to explore the solution space and to discover a set of routes that are stored in a pool. The last component recombines the stored routes using a Set Packing formulation to produce good quality solutions. Cura (2014) solves the TOPTW using an Artificial Bee Colony (ABC) approach that mimics the foraging behavior of honey bees. He introduced a new food source acceptance criterion based on SA and a new scout bee search behavior based on a local search procedure. Guibadj and Moukrim (2014) designed a Memetic Algorithm (MA) that uses the *route-first cluster-second* approach to solve the TOPTW. Solutions are represented as permutations of a subset of reachable customers called *giant tours*, and a splitting procedure is applied to extract the optimal set of feasible routes with respect to the order of visits in the giant tour. In Gunawan et al. (2015c), the authors extend the ILS algorithm of Gunawan et al. (2015a) to solve the TOPTW. The algorithm generates an initial solution using a greedy insertion heuristic that chooses the customers to be inserted based on roulette-wheel selection. The initial solution is then improved using an ILS that uses different local search operators and a combination of acceptance criteria and perturbation mechanisms to balance between diversification and intensification in the search process. In a later work, Gunawan et al. (2015b) presented a hybridization that embeds an ILS into a SA algorithm to address the drawback that is the early termination of the ILS. Schmid and Ehmke (2017) proposed an "Effective Large Neighborhood Search" (ELNS) approach to solve the TOPTW. The algorithm starts by generating an initial solution. During each iteration of the algorithm, the solution is destroyed and repaired through operators specifically designed for the TOPTW. To the extent of our knowledge, the ELNS is currently the approach that provides the largest proportion of best-known solutions for the TOPTW benchmark instances.

Finally, for a more detailed overview of the literature about orienteering problems, we invite the reader to refer to Vansteenwegen et al. (2011) and Gunawan et al. (2016) who provide reviews on several relevant variants of the orienteering problem, including the TOPTW, and discuss applications and solution methods, both exact and heuristic, for OP variants.

3. Problem definition

The TOPTW is defined on a complete directed graph $G = (V, A)$, where $V = \{0, 1, 2, ..., n\}$ is the vertex set, and $A = \{(i, j) : i \neq j,\ i, j \in V\}$ the arc set. Vertex 0 represents the depot, which corresponds to the starting and ending points, and each vertex $i \in V \setminus \{0\}$ represents a customer associated with a non-negative profit (score) p_i, a non-negative service time σ_i, and a predefined time window $[e_i, l_i]$. The profit of each customer $i \in V \setminus \{0\}$ can be collected at most once by a vehicle within its associated time window, i.e., a vehicle cannot visit the customer i if it arrives later than l_i, and in the event that it arrives earlier than e_i, it has to wait until e_i before the service can start. A time window $[e_0, l_0]$ is associated with the depot, where $e_0 = 0$ refers to the earliest departure time and l_0 refers to the latest possible arrival time at the depot. A non-negative travel time $c_{i,j}$ is associated with each arc $(i, j) \in A$. Travel times are assumed to satisfy the triangle inequality.

A fleet of m identical vehicles is available at the depot, and each of them performs at most one route. A route is an ordered subset of customers that are visited by the same vehicle. Routes must start and end at the depot. We note the total profit collected during a route r, $P(r) = \sum_{i=1}^{i=size(r)} p_{r[i]}$, and its total duration (travel cost) $C(r) = c_{0,r[1]} + \sum_{i=2}^{i=size(r)} (c_{r[i-1],r[i]} + W_{r[i]} + \sigma_{r[i]}) + c_{r[size(r)],0}$, where $r[i]$ denotes the i^{th} customer of the route and W_u the vehicle's waiting time at customer u. Note that this travel cost corresponds to the arrival time at the depot. The total duration of each route is constrained within a predefined time limit L_{max}. We assume that L_{max} equals the latest possible arrival time at the depot, i.e., $L_{max} = l_0$. A route is considered feasible if and only if $C(r) \leq L_{max}$ and if each customer is visited within its time window. Thus, a feasible TOPTW *solution* S consists of at most m feasible routes in which each customer is visited at most once. The objective is to find a solution S that maximizes the total collected profit, i.e., that maximizes $\sum_{r \in S} P(r)$. For mixed integer linear programming formulations of TOPTW see (Montemanni and Gambardella, 2009; Vansteenwegen et al., 2009).

4. Solution approach

4.1. Method overview

In this paper, we propose a randomized Multi-Start Iterated Local Search procedure (MS-ILS) enhanced by an adaptive memory mechanism. Originally, MS procedures are simple memoryless algorithms that sample the solution space by applying a local or neighborhood search from multiple randomly generated initial solutions. If an initial solution falls inside the attraction basin of a global optimum, the local search

will pull it to this global optimum. Due to their simplicity, more often than not, MS procedures alone have difficulties to compete with more aggressive metaheuristics and must be strengthened by complementary diversification techniques to help surmount local optima.

Our approach employs an Iterated Local Search (ILS) procedure as the local search step of the MS metaheuristic. The ILS is a simple metaheuristic whose principle is to build a sequence of improved local optima to explore the search space. More specifically, starting from a solution S, at each iteration, the ILS generates a new solution S' by perturbation of S, then improves it using a local search to obtain S''. If S'' satisfies an acceptance criterion, it becomes starting solution for the next iteration; otherwise the algorithm goes back to S.

The efficiency of our algorithm stems from two key aspects. The first one is to alternate between two search spaces, each one using a different solution representation. This idea has proven to be very effective for various vehicle routing problems (Prins, 2004; Prins et al., 2009; Mendoza and Villegas, 2013; Montoya et al., 2016), including the Team Orienteering Problem (TOP) (Bouly et al., 2010; Dang et al., 2013). In our case, the two solution representations are the *route representation* and the *giant tour representation*. The *route representation* is a genuine TOPTW solution, i.e., a set of feasible routes, one for each vehicle in use. To ensure fast feasibility checks upon insertion of a customer, we record, for each route, additional information on the arrival time, the waiting time and the maximum delay allowed for the service at each customer. On the other hand, the *giant tour representation* consists of an ordered list T of all the accessible customers in V with no route delimiters. It is a permutation $T = (T[1], T[2], ..., T[n])$ of n customers that ignores route length constraints and time windows. The interesting part about this indirect solution representation is (1) the fact that one giant tour corresponds to multiple TOPTW solutions, and (2) that it is possible to retrieve the optimal solution with respect to the order of customers in polynomial time using a *splitting* procedure. Thus, it is possible to search the space of giant tours instead of the TOPTW solution set, without loss of information. The reverse transformation from the *route representation* to the *giant tour representation* is achieved through a simple concatenation procedure.

The second key component is the integration of a long-term memory mechanism in order to improve the performance of the algorithm by learning from local optima found during previous iterations. One important drawback of a pure multi-start procedure is being a memoryless method: each iteration is independent of the previous one and no information about the solutions is passed from one iteration to the other. In our approach, we integrate an adaptive memory mechanism to retain information about interesting customer sequences observed in high-quality solutions. The use of a memory mechanism is inspired from the probabilistic tabu search of Rochat and Taillard (1995). It consists in storing routes extracted from high-quality solutions inside a pool of elite but diverse routes, and then using them to generate new starting solutions. In our approach, the stored routes are used to construct giant tours using a probabilistic function that is biased towards the selection of elements that appear more frequently in high-quality solutions.

Algorithm 1 describes the general structure of our MS-ILS approach. At the beginning, an iterative

Algorithm 1: MS-ILS algorithm for the TOPTW

Data: TOPTW instance;

$\mathcal{M}$: adaptive memory;

$iter_{ILS}$: number of iterations of the ILS;

$iter_{max}$: maximum number of restarts.

Result: S_{best} overall best solution for the TOPTW instance.

```
1  begin
2      iter ← 0
3      (M, S_best) ← initializeMemory()                    // see Algorithm 2
4      repeat                                              // Multi-start loop
5          T ← constructGiantTour(M)
6          S ← split(T)                                    // see Section 4.5
7          for j ← 1 to iter_ILS do                        // ILS loop
8              T' ← randomRotation(T)                      // Perturbation
9              S' ← split(T')
10             S' ← localSearch(S')                        // see Section 4.6
11             updateMemory(M, S')                         // see Section 4.3
12             if f(S') ≥ f(S) then
13                 T ← concat(S')
14                 S ← S'
15         if f(S) ≥ f(S_best) then S_best ← S
16         if new routes have been added into M then       // see Section 4.3
17             iter ← 0
18         else
19             iter ← iter + 1
20     until iter = iter_max
21     return S_best
```

heuristic (described in Section 4.2) is called to initialize the adaptive memory $\mathcal{M}$ and the overall best solution S_{best}. At each iteration of the main loop, the algorithm constructs a giant tour T using the routes stored inside the adaptive memory $\mathcal{M}$ and extracts the solution S associated with tour T using the *split* procedure described in Section 4.5. It then performs an ILS (ILS loop) with T as the starting point. The ILS loop is executed until the maximum number of iterations $iter_{ILS}$ is reached. At each iteration of the ILS loop, T is perturbed by applying a random rotation, and a new giant tour T' is derived. The perturbation procedure considers T as a circular array and shifts its elements by a random number of positions. Afterwards, the associated solution S' is extracted from T', improved using the local search procedure described in Section 4.6, and the obtained routes are inserted into $\mathcal{M}$ if they are of good enough quality; otherwise they are ignored. If S' is better than S, it is concatenated into a new giant tour which replaces T in the next ILS iteration, and S is updated. After the ILS loop, S_{best} is updated and the number of main loop iterations *iter* is reset to 0. Otherwise, *iter* is incremented and the main loop restarts. The algorithms stops when it fails to insert new routes in the adaptive memory during $iter_{max}$ consecutive iterations.

4.2. Memory initialization

The initial set of routes that compose the adaptive memory is generated using a fast heuristic procedure that relies on a Best Insertion Algorithm (BIA) to build feasible solutions. At each iteration, the BIA evaluates all the feasible insertions of unrouted customers then selects and carries out the best one.

The insertion of a customer u between two successive nodes i and j is feasible if and only if u is visited within its time window, the visits to j and to the subsequent nodes remain within their respective time windows, and the length of the route remains less than L_{max}. Note that i and j can be either customers or the depot. Similar to Vansteenwegen et al. (2009), in order to ensure that the feasibility of an insertion is checked in $O(1)$, we record, for each customer included in the solution, its waiting time W_i and the maximum duration by which its service can be delayed without rendering the route infeasible $MaxShift_i$. These two measures are given by the following formulas:

$$W_i = \max\{0, e_i - a_i\}$$
$$MaxShift_i = \min\{l_i - (a_i + W_i), W_j + MaxShift_j\}$$

where a_i is the arrival time at node i, $[e_i, l_i]$ the time window associated with node i, and j is the node visited after i in the solution. The insertion of u between the nodes i and j delays the arrival at node j by $Shift_{u,i} = (c_{i,u}+W_u+\sigma_u+c_{u,j}-c_{i,j})$. The insertion of u is feasible if $a_u \leq l_u$ and $Shift_{u,i} \leq W_j+MaxShift_j$.

Fig. 2, illustrates a route containing four customers. The maximum length of the route is $L_{max} = 40$. For each visited customer, we display information on its time window, arrival time, waiting time, and the maximum duration by which its service can be delayed. Customer u can be inserted between customers 2 and 3 since $Shift_{u,2} = (c_{2,u} + W_u + \sigma_u + c_{u,3} - c_{2,3}) = 6 \leq 12 = W_3 + MaxShift_3$.

For each feasible insertion, the following criterion is calculated:

$$z_{u,i} = Shift_{u,i}/(p_u)^{\alpha}$$

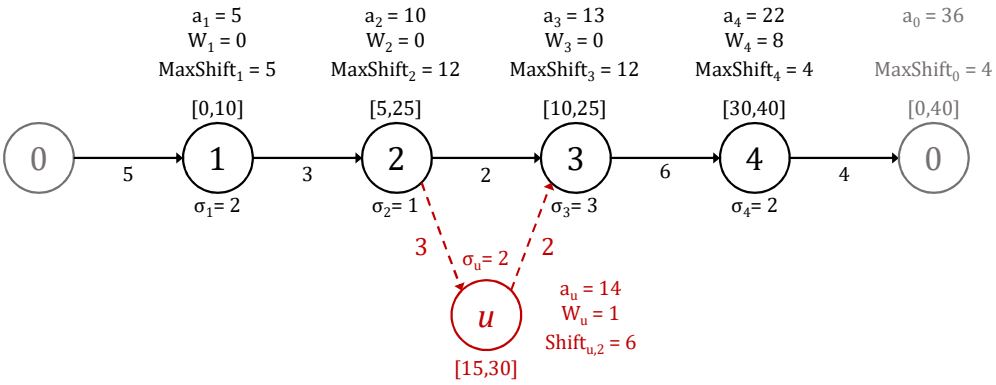


Figure 2: Feasibility check.

where α is a control parameter. The best insertion is determined by the couple (u, i) for which the criterion $z_{u,i}$ is minimized. The aim of this criterion is to favor the insertions which offer a good trade-off between the increase of the tour length and the collected profit. The parameter α is used to prioritize profit over the time consumption of an insertion. The value of α is discussed in Section 5.2. After the insertion of u between i and j, the information associated with the successors of u in the route are updated using the following formulas:

$$W_j^* = \max\{0, W_j - Shift_{u,i}\}$$
$$Delay_j = \max\{0, Shift_{u,i} - W_j\}$$
$$MaxShift_j^* = MaxShift_j - Delay_j$$

where $Delay_j$ represents the amount of time by which the service of customer j is delayed. When updating the data of a visit after j, $Shift_{u,i}$ is replaced with the delay of the visit that precedes it.

A pseudo-code of the initialization heuristic is given in Algorithm 2. First, a feasible solution is constructed from scratch using the BIA described above. After that, at each iteration of the heuristic, a sequence of consecutive customers is removed from each route of the solution. The solution is then rebuilt by inserting unrouted customers using the BIA. The sequences to remove are identified by a starting position *start* and a length of q customers. The length of the removed sequences q is first set to 1, and at the end of each iteration, *start* is moved by q positions and q is incremented by one. If the solution is improved after repair, q is reset to its initial value. Destruction is applied in a circular manner, i.e., when the end of a route is reached, the removal continues from its beginning. This part of the algorithm is similar to the *parallel remove-and-repair* operator described in Section 4.6. In order to keep q from becoming too big and causing a large part of the solution to be destroyed at each iteration, it is reset to 1 each time its value reaches half the number of routed customers of the largest route (r_{max}) in the solution, in terms of the number of visited customers. The destruction and construction phases are repeated until the algorithm reaches $iter_{init}$ iterations without improving the best solution.

Algorithm 2: Memory initialization algorithm

Data: V: a vector of n customers

Result: S_{best}: overall best solution

$\mathcal{M}$: an adaptive memory containing a set of routes

```
1  begin
2      S ← applyBIA({}, V)
3      iter ← 0
4      start ← 0
5      q ← 1
6      while iter < iter_init do
7          S ← applyParallelDestruction(S, start, l)
8          U ← getUnroutedCustomers(V, S)
9          S ← applyBIA(S, U)
10         updateMemory(M, S)                          // see Section 4.3
11         if f(S) ≥ f(S_best) then
12             S_best ← S
13             iter ← 0
14             q ← 0
15         else
16             iter ← iter + 1
17         start ← start + q
18         q ← q + 1
19         if q ≥ size(r_max)/2 then q ← 1
20     return (M, S_best)
```

4.3. Memory update

For performance reasons, we limit the size of the adaptive memory to $Size_{\mathcal{M}}$. When it becomes full, we need to remove some routes to be able to insert new ones. In order to keep promising routes, we apply the following strategy. Before its insertion, each route is labeled with the total collected profit and the total duration of the solution to which it belongs. The adaptive memory is then sorted in decreasing order of profits using the duration to resolve any equality. When the adaptive memory is full, the route to be inserted is first compared to the weakest route of the memory. If the new route's label is weaker, i.e., it has a lower total collected profit, the route is ignored. Otherwise, it is added to the memory and the weakest route is deleted. The idea is that routes that belong to solutions with higher scores are more likely to contain elements of optimal or sub-optimal solutions. To speed up the initialization of the adaptive memory, the routes of each

solution generated by the BIA are considered for inclusion in the memory, so the heuristic is only used once.

4.4. Giant tour construction

The construction of giant tours plays an important role in our algorithm since it allows us to use information collected through past iterations for diversification and intensification purposes. To construct a giant tour, we select m routes from the adaptive memory and combine them as follows. Let $\mathcal{M}'$ be a copy of the current memory. First, we extract a route r from $\mathcal{M}'$ in a probabilistic way, and discard from $\mathcal{M}'$ all the routes r' that share at least one common customer with r. This process is then repeated until m routes are extracted, or until $\mathcal{M}'$ becomes empty. When selecting routes, we favor those extracted from solutions with higher quality. To that effect, we use the roulette wheel selection mechanism. After that, the depot is removed from each of the selected routes. The resulting routes are then concatenated into a random order to form a partial giant tour, which we complete by randomly spreading out the unrouted customers over the tour. These unrouted customers are randomly inserted at the beginning of the tour, at the end, or between the routes in such a way that the m selected routes remain untouched, as can be seen in the example depicted in Fig. 3.

4.5. Splitting algorithm

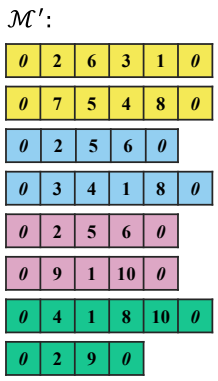
In this section, we present the *optimal splitting algorithm* used to extract a TOPTW solution from a *giant tour*. Splitting a *giant-tour* T consists of extracting m distinct sub-sequences from T such that, (1) each extracted sub-sequence corresponds to a feasible TOPTW route, (2) there are no shared customers between the extracted sub-sequences, and (3) the total profit of the extracted sub-sequences is maximized. In the remainder of this section, we will refer to the above sub-sequences of T by *extracted routes* or simply *routes*, for convenience.

When splitting a *giant-tour* T, not all the feasible sub-sequences of T need to be considered to select m routes that maximize the total profit. Bouly et al. (2010) introduced the notion of *saturated routes* and showed that solutions containing only *saturated routes* are dominant when splitting giant-tours for the TOP. In the following, we provide an adapted definition of *"saturated routes"*, and show that the same dominance property holds when splitting giant-tours for the TOPTW.

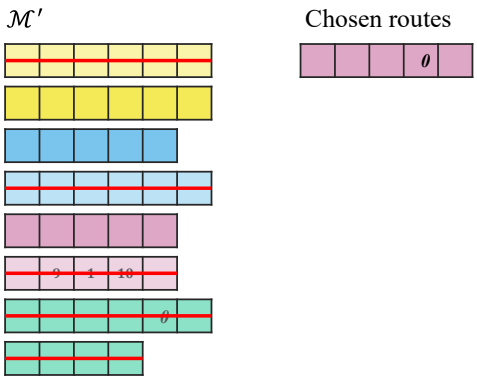
Definition 1. *Given a giant tour* $T = (T[1], T[2], \dots T[n])$*, a "saturated route" in* T *is a sub-sequence* $\pi_i = (T[i], T[i+1], \dots T[j])$*,* $1 \leq i \leq j \leq n$*, such that the route* $r = (0, T[i], T[i+1], \dots, T[j], 0)$ *is feasible and either* $j = n$*, or the route* $r' = (0, T[i], T[i+1], \dots, T[j], T[j+1], 0)$ *is infeasible.*

Proposition 1. *For any instance of the TOPTW where* m *is the number of available vehicles, for any giant-tour* T *of this instance, there exists an optimal solution to the splitting problem of* T *where all the extracted routes are saturated.*

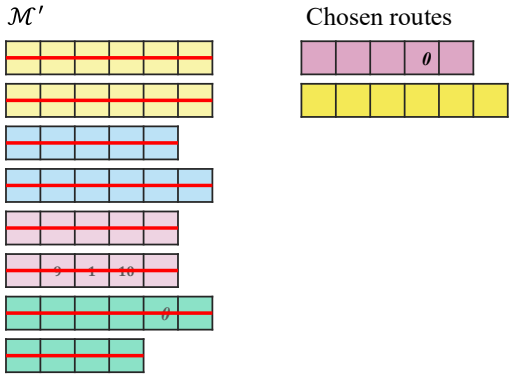
Proof. Let T be a giant-tour visiting all the customers of a TOPTW instance in any order, and let S be an optimal solution obtained when T is split into m routes. S contains at most m extracted routes



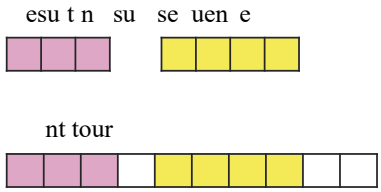
(a) Adaptive memory containing routes from 4 different solutions ($n = 10$ and $m = 2$).



(b) Choose one route and discard all that share at least one customer with it from the adaptive memory.



(c) Choose a second route the same way as in Fig. 3b.



(d) Arrange the chosen routes and complete the giant tour by inserting the remaining customers at random.

Figure 3: Illustration of the giant tour construction process.

$\pi_{i_k} = (T[i_k], T[i_k + 1], \ldots, T[j_k]),\ 1 \leq k \leq m,\ 1 \leq i_k \leq j_k \leq n$ which, can be re-ordered in such a way that $1 \leq i_1 \leq j_1 < i_2 \leq j_2 < \cdots < i_m \leq j_m \leq n$. Let π_{i_k} be the k-th route in S. Two cases are identified:

- Case 1 : route π_{i_k} is such that $j_k + 1 < i_{k+1}$ (i.e. there are unrouted customers between the end of π_{i_k} and the start of $\pi_{i_{k+1}}$). If π_{i_k} were not saturated, then route $\pi'_{i_k} = (T[i_k], T[i_k + 1], \ldots, T[j_k], T[j_k + 1])$ would be feasible, would have a greater profit, and would not conflict with the other extracted routes. As such, it would be possible to improve S by replacing π_{i_k} with π'_{i_k}. This contradicts the hypothesis that S is optimal. Hence, π_{i_k} is saturated.

- Case 2 : route π_{i_k} is such that $j_k + 1 = i_{k+1}$ (i.e. route $\pi_{i_{k+1}}$ starts immediately after route π_{i_k} ends). If π_{i_k} is not saturated, then route $\pi'_{i_k} = (T[i_k], T[i_k + 1], \ldots, T[j_k], T[i_{k+1}])$ would be feasible, and route $\pi'_{i_{k+1}} = (T[i_{k+1} + 1], T[i_{k+1} + 2], \ldots, T[j_{k+1}])$ would also be feasible, due to the triangle inequalities. Furthermore, π'_{i_k} and $\pi'_{i_{k+1}}$ do not share customers with one another, and the sum of their profits is equal to that of π_{i_k} and $\pi_{i_{k+1}}$. Thus, by replacing π_{i_k} and $\pi_{i_{k+1}}$ with π'_{i_k} and $\pi'_{i_{k+1}}$, respectively, in S, we obtain a new optimal S' with one less *unsaturated* route. Hence, by reiterating the process on every

unsaturated route of S, we can produce an optimal solution where all the routes are saturated.

□

Dang et al. (2013) presented an evaluation procedure that efficiently uses the limited number of saturated routes to reduce the complexity of the splitting process. They reduced the splitting problem to a knapsack problem with conflicts (KPC) (Takeo et al., 2002), and used a dynamic programming algorithm to determine the optimal splitting in polynomial time. In this paper, we extend this splitting procedure to tackle time window constraints.

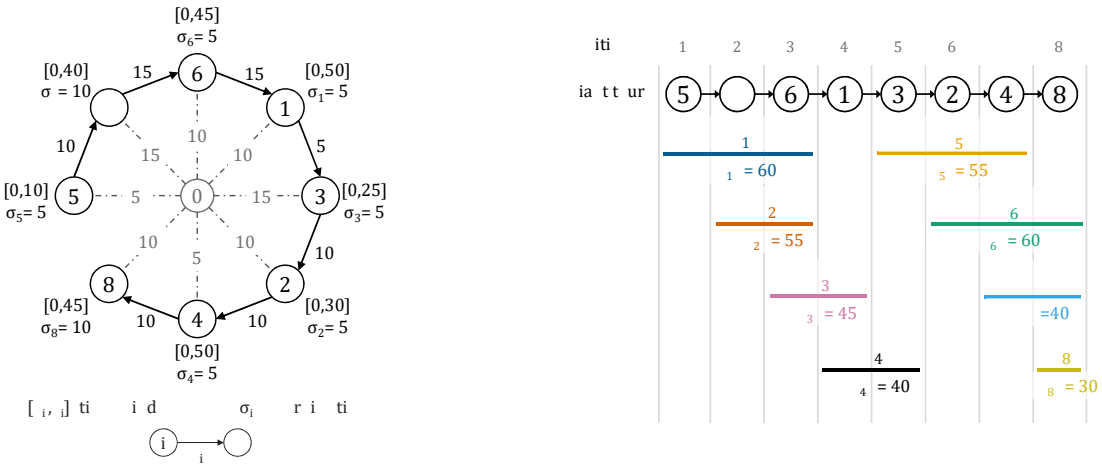
4.5.1. Extraction of saturated routes

When splitting a *giant-tour*, we extract one *saturated route* starting from every customer in the tour. Let $\Pi_T = \{\pi_1, \pi_2, ..., \pi_n\}$ denote the set of the n possible saturated routes extracted from the giant tour T. When extracting these routes, it must be ensured that each customer is visited within its time window and that the route is completed within the given time limit L_{max}, i.e., that $C(\pi_i) \leq L_{max}\ \forall i \in \{1, ..., n\}$. Starting from customer $T[i]$, we initialize the route $\pi_i = (0, T[i], 0)$. We then extend it by consecutively including the following customers $T[j], j \geq i$ as long as they are visited within their time windows. If the vehicle arrives at a customer $T[j]$ before the beginning of its time window, a waiting time is added. If the vehicle arrives too late at customer $T[j]$, if the inclusion of $T[j]$ violates the time limit constraint, or if the process reaches the end of the giant tour, the extension of π_i is stopped and the route is considered saturated. In the split algorithm for the basic TOP, all the saturated routes can be extracted in $O(n)$ because the cost of any sub-sequence $(i+1, ..., j, j+1)$ can be deduced in $O(1)$ from that of $(i, ..., j)$. However, this property is not satisfied by TOPTW sub-sequences. In our best implementation, route extraction runs in $O(n^2)$.

Fig. 4 shows a giant-tour for a TOPTW instance with 8 customers, and all the saturated routes that can be extracted from it. The maximal length of a route is $L_{max} = 60$. The giant-tour is represented in Fig. 4a. It is a permutation of the 8 customers. The figure displays the travel time between every two successive nodes, and gives for each node, its service time σ_i, its time widow (between brackets), and its travel time to and from the depot. The saturated routes are presented in Fig. 4b. Route π_1, for example, corresponds to the route $(0, 5, 7, 6, 0)$ and has a length $C(\pi_1) = 5 + 5 + 10 + 10 + 15 + 5 + 10 = 60$. Customer 1 cannot be included in this route, since its inclusion makes the total travel time greater than L_{max}. In the case of route π_3, customer 3 cannot be included in the route, not because of the time limit, but because the vehicle reaches it at instant 35, when its time window has already closed. Route π_8 consists only of customer 8, because it is the last customer of T.

4.5.2. Selection of saturated routes

Once all the saturated routes have been extracted from T, the problem of splitting T into a feasible TOPTW solution becomes that of choosing m routes that do not share customers and that maximize the total collected profit. This can be formulated as a *Knapsack Problem with Conflicts* (KPCG).



(a) Giant-tour for a TOPTW instance with $n = 8$ and $L_{max} = 60$.

(b) Saturated routes extracted from T.

Figure 4: Extraction of saturated tours.

In a KPCG, we consider a set of items to be put into a knapsack of limited volume. Each item is associated with a value and a volume, and some items are in conflict with each other. Conflicting items cannot be put in the knapsack together. The objective of the KPCG is to find a subset of non-conflicting items that fits into the knapsack and maximizes the total packed value. In our splitting procedure, the volume of the knapsack is equal to m, the maximal number of available vehicles. Each *saturated route* π_i corresponds to an item i, and two items i and j are in conflict if their corresponding routes π_i and π_j share customers.

Usually, conflicts between items in a KPCG are modeled using a graph, called a *conflict graph*. In our particular case, the conflict graph for the splitting problem is an *interval graph* (Tarjan, 1975). In short, a graph $H = (X, U)$ is an interval graph if there is a mapping I between the vertex set X and sets of consecutive integers (called *intervals*) such that two vertices in X are adjacent if and only if their respective corresponding intervals intersect i.e. $\forall i \in X, \forall j \in X,\ (i,j) \in U \iff I(i) \cap I(j) \neq \emptyset$. To solve the KPCG, we use the algorithm proposed by Sadykov and Vanderbeck (2013). They devised a pseudo-polynomial algorithm to solve a KPCG with interval conflict graphs, defined with n items and a knapsack capacity equal to W, in $O(n \cdot W)$. Hence, the following result for the TOPTW emerges.

Proposition 2. *Given a TOPTW instance with m available vehicles and n saturated tours extracted from a giant tour T, the splitting of T can be done optimally in $O(m \cdot n)$ time and space.*

Proof. Each saturated route extracted from a *giant-tour* T can be represented by the set of positions it covers, i.e., each route $\pi_i = (T[i], T[i+1], \ldots, T[j]),\ 1 \leq i \leq j \leq n$ is represented by the set $I(\pi_i) = \{i, i+1, \ldots, j\}$ of consecutive integers. Hence, the set of saturated routes can be mapped to the set of vertices of an interval graph H. Moreover, a non-empty intersection between two intervals $I(\pi_i)$ and $I(\pi_{i'})$ indicates the presence of shared customers between the two routes π_i and $\pi_{i'}$, and corresponds to an edge (i, i') in the graph H. As outlined above, once the saturated routes are extracted, the problem of splitting a giant-tour becomes

that of choosing m saturated routes that do not share customers and such that the total collected profit is maximized. This translates into solving a KPCG where the volume of the knapsack is equal to m, all the items have unitary weights and their values are equal to the profit of the corresponding route. The conflict graph H associated to this KPCG is an interval graph. Given the capacity m of the knapsack and the number of saturated routes n, and based on the work of Sadykov and Vanderbeck (2013), we conclude that the splitting can be done in $O(m \cdot n)$ time and space. □

Figure 5 illustrates the splitting process of the giant-tour presented in Fig. 4. We want to split T into a solution with two routes ($m = 2$) with $L_{max} = 60$. Fig. 5b shows the saturated routes that were extracted from T and their collected profit $P(\pi_i)$. Observe that each saturated route corresponds to a sub-interval of the interval $[1, 8]$. For example, route π_1 corresponds to the interval $[1, 3]$ (as it covers the first three nodes of T), route π_3 corresponds to the interval $[3, 4]$, and route π_8 corresponds to $[8, 8]$. Furthermore, if two routes share at least one customer, then the intersection of their intervals is not empty, and vice versa. Fig. 5c represents the interval graph that corresponds to the conflict graph of the saturated routes. Every vertex of the graph is mapped to the interval of positions of the corresponding saturated route, and two vertices are adjacent if and only if their respective intervals intersect with each other. Finally, the optimal solution obtained after solving the KPCG is shown in Fig. 5d. It is composed of the two routes π_3 and π_6 and its total score is 155.

4.6. Local search

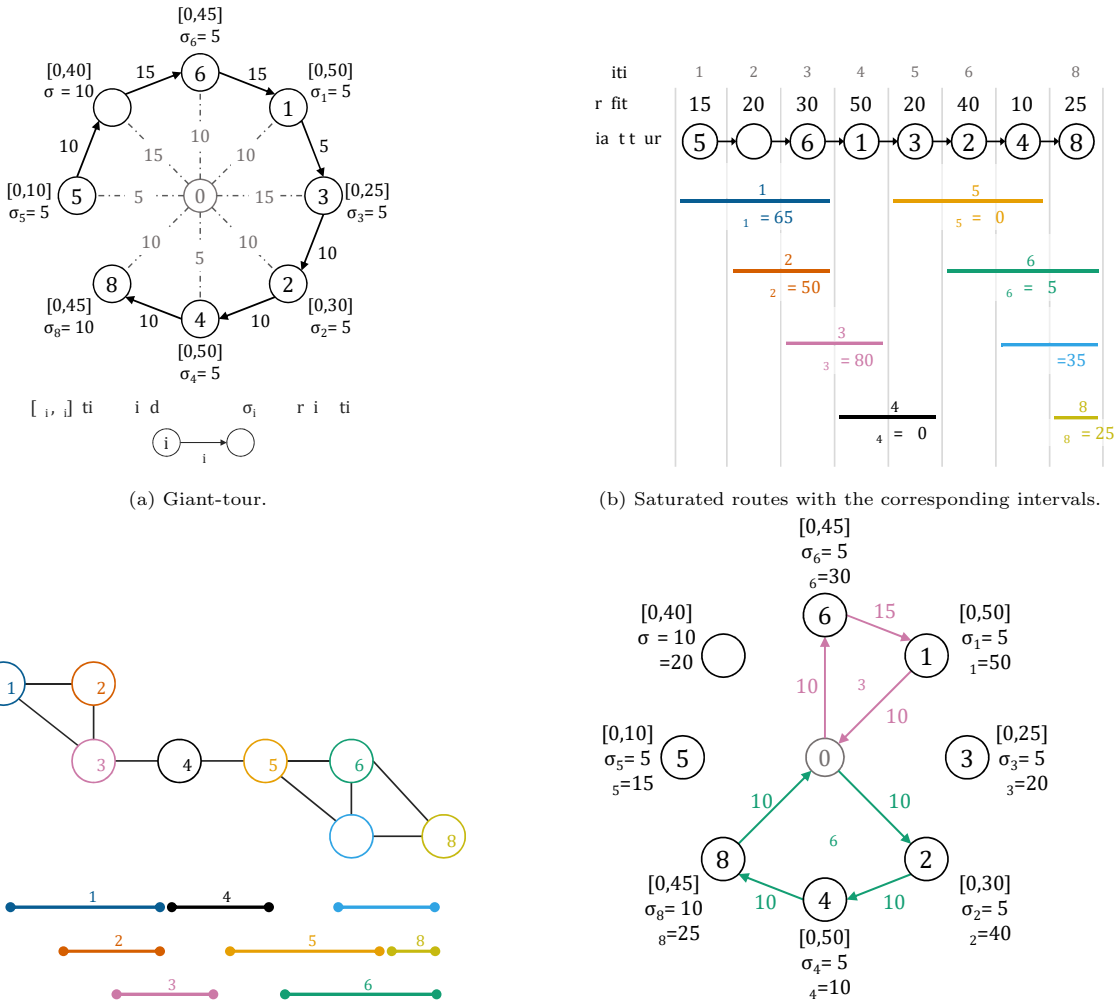
The local search (LS) procedure is a key component of our algorithm since it serves as an intensification mechanism to improve the quality of the solutions constructed at each iteration. As previously mentioned, our algorithm uses two different solution representations, each with its own characteristics. Hence, we divide the neighborhood operators that compose the local search procedure into two sets: the first set is applied on *giant tours*, while the second set is performed on *routes*.

4.6.1. Giant tour neighborhoods

The LS includes two neighborhoods for *giant tours*:

- *shift operator*: moves a randomly chosen customer from its current position in the *giant tour* to a different one;
- *swap operator*: chooses a random customer from the *giant tour* and exchanges its position with that of another randomly chosen customer;

Because the routes that compose a solution are not explicitly defined in the *giant tour representation*, the *shift* and *swap* operators do not need to check if feasibility is maintained, since every move is considered feasible in a giant tour representation. All that is needed to evaluate a move is to extract the new routes by splitting the giant tour. For performance purposes, we use the greedy splitting algorithm described in greater detail below.



(a) Giant-tour.

(b) Saturated routes with the corresponding intervals.

(c) Conflict graph for the KPCG.

(d) Saturated optimal solution.

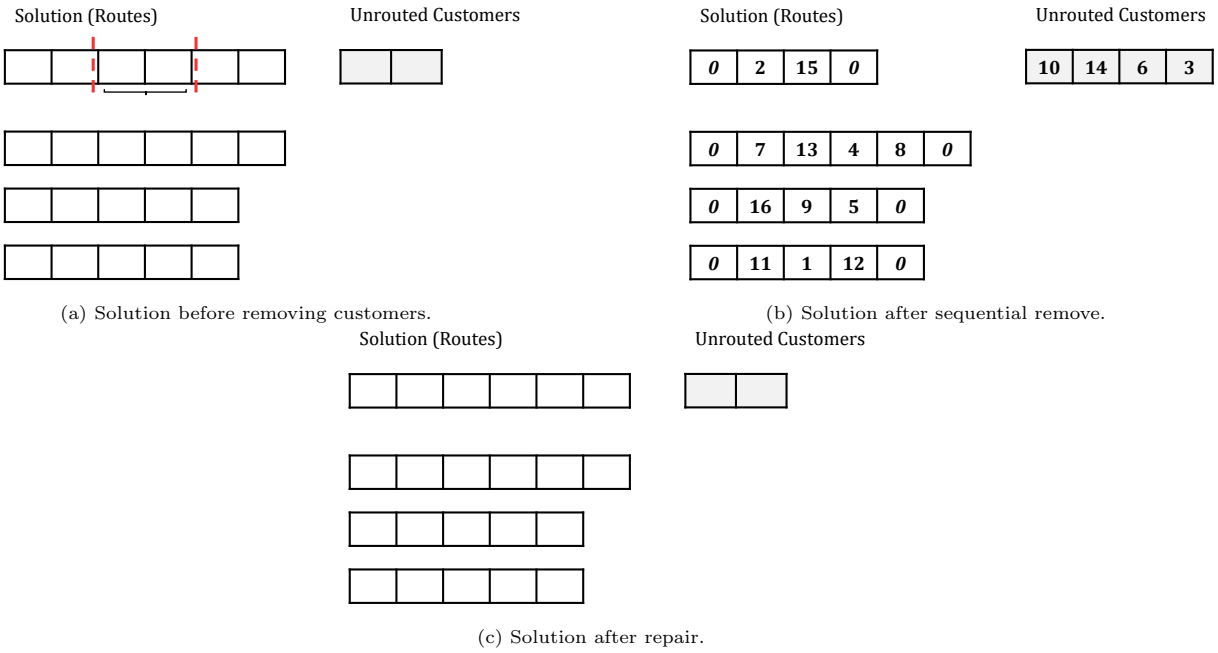
Figure 5: Example of the splitting procedure for a TOPTW instance with $m = 2$, $n = 8$, and $L_{max} = 60$.

4.6.2. Route neighborhoods

For *route* improvement, the LS procedure uses the classical routing neighborhoods *Or-Opt* and *2-Opt** to reduce the total travel time of each route, and three other neighborhoods to fill the resulting time saved in order to increase the total profit. The *2-Opt** operator replaces arcs $(i, i+1)$ from route u and $(j, j+1)$ from route v with arcs $(i, j+1)$ and $(j, i+1)$, thus interchanging the two sub-paths without altering the order of visits. The *Or-Opt* operator relocates a sequence of one or two successive customers in the same route, without altering their order of visit. The remaining operators follow the same *remove-and-repair* principle of the initialization heuristic described in Section 4.2 but differ in the way they select the customers to be removed and in how they repair the solution:

- *Random remove-and-repair*: removes d randomly chosen customers from the solution and, then, using the Best Insertion Algorithm (BIA), inserts currently unrouted customers. At each iteration, d is randomly generated in the interval $[1, D_{max}]$.

- *Sequential remove-and-repair*: this operator focuses on improving one single route at a time. This operator removes from each route r a sequence of q consecutive customers starting from position $start$ mod $size(r)$ and uses BIA to insert new customers into the destroyed route. At each iteration, $start$ is moved by q customers, and q is increased by one. When all the customers in a route have been removed at least once, it skips to the next route in the solution. This operator stops if an improvement is found, or once all the routes of the solution have been tested. Fig. 6 shows an iteration of this operator.



(a) Solution before removing customers.

(b) Solution after sequential remove.

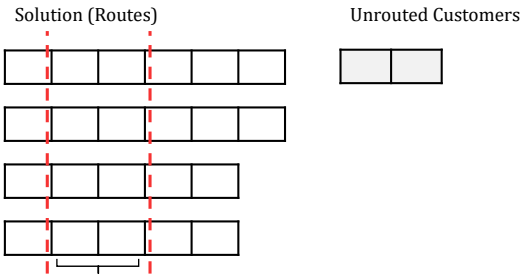
(c) Solution after repair.

Figure 6: Sequential remove-and-repair.

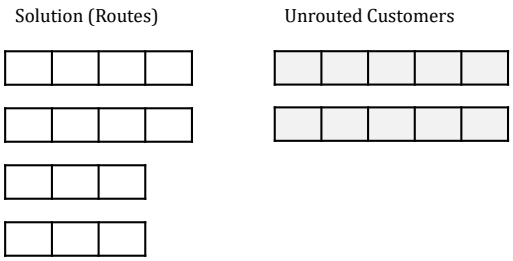
- *Parallel remove-and-repair*: Similar to the previous operator, this operator removes from each route r a sequence of q consecutive customers starting from position $start$ mod $size(r)$. However, instead of focusing on one route, it is applied in parallel on all the routes of the solution, and then inserts unrouted customers using the BIA. An example of this move is shown in Fig. 7. The search in this neighborhood stops when an improvement is found, or when each customer in the solution has been removed at least once. Removing parallel sequences from different routes creates free time slots across the whole solution and gives the opportunity to BIA to move customers between routes, and introduce more profitable ones in the solution.

4.6.3. Local search algorithm

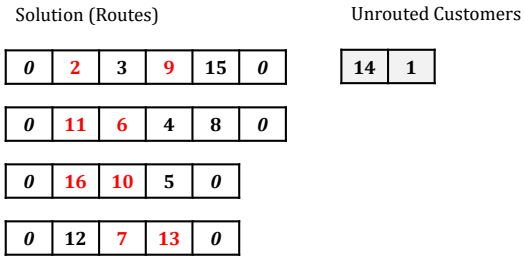
The local search procedure works as follows. At each iteration, the LS randomly selects one of the previous neighborhoods and explores it. All the neighborhoods have the same probability of being chosen whether they consider *routes* or *giant tours*. The search in a given neighborhood stops as soon as an improvement is found



(a) TOPTW solution.



(b) Solution after parallel remove.



(c) Solution after repair.

Figure 7: Parallel remove-and-repair.

or when no improving move can be found. When a neighborhood operator fails to find an improvement, it is discarded until the solution is improved by another operator. This process is repeated until all neighborhoods fail to find an improvement to the current solution.

4.6.4. Greedy split and concatenation

Because the LS procedure uses two different sets of neighborhoods, and because it can randomly switch from an operator of one set to that of the other, the number of times a solution shifts from one representation to the other can grow large very quickly. This negatively impacts computation times, mainly due to the optimal splitting procedure. To address this aspect of the LS, we devise a fast heuristic splitting method and a giant tour construction process to go with it.

Inside the LS algorithm, the construction of a *giant tour* T is achieved by concatenation of the routes that compose the solution one after the other, and then by appending the remaining unrouted customers at the end of the tour.

On the same principle as the above fast concatenation method, we describe a greedy splitting heuristic to evaluate a giant tour $T = (T[1], T[2], ..., T[n])$. Consider $\theta_k = (T[i], T[i+1], ..., T[n])$ a sub-sequence of T. It is possible to divide θ_k into two parts by extracting the saturated route $\pi_{i_k} = (T[i_k], T[i_k+1], ..., T[j_k])$ and considering the second part $\theta_{k+1} = (T[j_k+1], T[j_k+2], ..., T[n])$ as a new sequence of unrouted customers. Our greedy splitting heuristic consists in performing the previous bisection m times on consecutive sequences θ_k, starting with $\theta_1 = T$. Fig. 8 shows an example of splitting using the heuristic procedure. The heuristic splitting is used as a quick way to evaluate moves in the *swap* and *shift* neighborhoods. The optimal splitting algorithm is applied at the end of the LS procedure to extract the best routes from the giant tour.

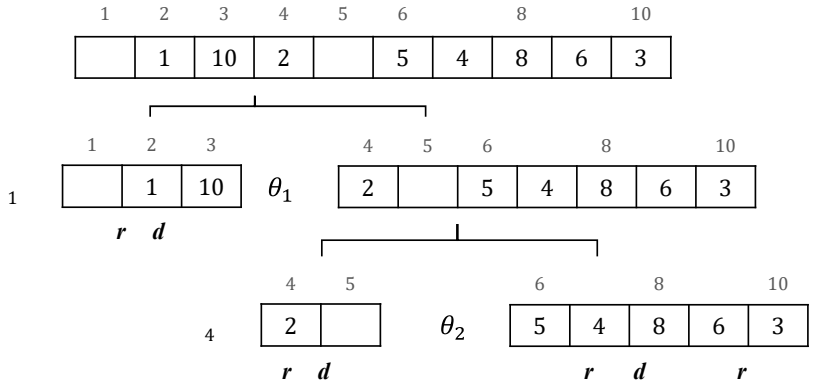


Figure 8: Heuristic splitting procedure when $m = 2$.

5. Computational experiments

Our algorithm was coded in C++ using the Standard Template Library (STL) for data structures, and compiled using the GNU GCC compiler in a Linux environment. In the following, we present the computational experiments that were carried out in order to tune the algorithm, to evaluate its performance relative to state-of-the-art algorithms, and to assess the contribution of each of its components. All the experiments were conducted on a single thread on an Intel Xeon X7542 CPU at 2.67GHz processor.

5.1. Benchmark instances

Our algorithm was tested on the various TOPTW benchmark instances. All the literature benchmarks are available online at `https://www.mech.kuleuven.be/en/cib/op`.

TOPTW benchmark instances are derived from Solomon et al.'s (Solomon, 1987) instances for the Vehicle Routing Problem with Time Windows (VRPTW), and Cordeau's (Cordeau et al., 1997) instances for the Multi Depot Periodic VRPTW (MDPVRPTW). The instances were adapted by considering the customer demands in the original data sets as node profits in the TOPTW instances. Travel time between two vertices is assumed to be equal to the euclidean distance. It is rounded down to the first decimal for Solomon's instances and to the second decimal for Cordeau's instances. Solomon's instances are organized into six sets: C100, C200, R100, R200, RC100 and RC200, divided according to the distribution of vertices on the plane (clustered, random, random-clustered) and the width of the time windows (narrow, wide). Each instance contains either 50 or 100 customers. Since Cordeau's instances were originally meant for a periodic variant of the VRP, when using them for the TOPTW, we consider that all customers are available on the same day. Instances in this data set contain a number of customers ranging from 48 to 288.

Righini and Salani (2009) constructed the first OPTW instances using 29 of Solomon's instances from sets C100, R100 and RC100, and ten instances from Cordeau's (pr1 to pr10). The first benchmark instances for the TOPTW were introduced by Montemanni and Gambardella (2009). They used the earlier OPTW instances for the case where $m = 1$ and extended them by increasing m to two, three, and four available vehicles. They also proposed another 27 instances from sets C200, R200, and RC200 from Solomon's instances, and ten

others from Cordeau's instances (pr11 to pr20). Furthermore, Vansteenwegen et al. (2009) used the original instances of Solomon and Cordeau to introduce new benchmark instances where the number of vehicles considered for each instance makes it possible to visit all the customers. Since it is possible to visit all customers, the optimal solution for these instances is known: it is equal to the sum of all customer profits. Like Hu and Lim (2014), we refer to these instances as the "OPT" data set. As Vansteenwegen et al. (2009) point out, algorithms to solve the TOPTW are generally not designed expecting the possibility of including all the customers, which makes solving some of the "OPT" instances to the optimum rather difficult.

5.2. Parameter tuning

As described in Section 4, our proposed MS-ILS has five sets of parameters:

- α, the control parameter of the BIA;
- D_{max}, the maximum number of customers removed by the random removal operator;
- $Size_{\mathcal{M}}$, the size of the adaptive memory;
- $iter_{init}$, the number of iterations of the initialization heuristic;
- $iter_{max}$ and $iter_{ILS}$, the number of iterations of the main algorithm.

In order to calibrate these parameters, we carried out preliminary experiments on a subset of randomly selected instances from the TOPTW instances described in Section 5.1. We started from base parameter values that we identified during the development of our MS-ILS and then tuned each parameter, one after the other, by varying the base value. For each parameter, we kept the best setting found before proceeding to tune the next parameter. Given that parameters $iter_{max}$, $iter_{ILS}$, and $Size_{\mathcal{M}}$ have a similar effect on the performance of the algorithm, we chose to tune them in a joint fashion. More details, on the tuning experiments can be found in Section A of the supplementary material associated with this paper.

The final calibration results are displayed in Table 1. The final parameter values were chosen to provide a good trade-off between solution quality and runtime from the authors' perspective.

Parameter	Description	Value
α	the control parameter of the BIA	$[1.8, 2.8]$
D_{max}	max. nb. of customers removed by the Random remove-and-repair operator	$0.25 * n_{routed}$
$Size_{\mathcal{M}}$	size of the adaptive memory	$80 * m$
$iter_{init}$	nb. of iterations of the initialization heuristic	1000
$iter_{max}$	nb. of iterations of the main algorithm	$2 * n/m$
$iter_{ILS}$	nb. of iterations of the ILS	4

Table 1: Final parameter values.

5.3. Computational comparisons

In order to evaluate the performance of our proposed MS-ILS for TOPTW, we compared its results with those of the following state-of-the-art algorithms:

- Variable Neighborhood Search (VNS) proposed by Tricoire et al. (2010),
- Greedy Randomized Adaptive Search procedure with Evolutionary Local Search algorithm (GRASP-ELS) of Labadie et al. (2011),
- LP-Based Granular Variable Neighborhood Search (GVNS) of Labadie et al. (2012),
- Iterative Three-Component Heuristic (I3CH) of Hu and Lim (2014).
- Large Neighborhood Search (ELNS) of Schmid and Ehmke (2017).

The results of GVNS, GRASP-ELS, I3CH, ELNS, and our MS-ILS were all obtained with five runs of the algorithm on each instance, while VNS was run ten times on each instance. Note that some results of VNS that were not originally reported by Tricoire et al. (2010) were made available on the authors' website $http://prolog.univie.ac.at/research/op/$. Since they are better results than those originally published, we used them instead for our comparisons.

For the sake of fair comparison between the algorithms in terms of computational effort, the reported running times of each algorithm were adjusted to account for the speed difference between the different computation setups. Similar to Hu and Lim (2014), we used the *Super Pi* benchmark for this purpose. *Super Pi* is a single-threaded program that computes the digits of π up to a specified number and is commonly used as an estimate of CPU speed. Table 2 indicates the CPU used by each algorithm, its *Super Pi* score, which corresponds to the number of seconds needed to compute the first one million digits of π, and the associated time scaling factor. To obtain this factor, we estimated the performance of each processor by considering the performance of our machine to be 1. For GRASP-ELS, GVNS and I3CH, we opted to use the *Super Pi* scores reported by Hu and Lim (2014). Unfortunately, the scores of VNS and ELNS are not available. Furthermore, we only had limited information on the experimental setup used by each of them to be able to estimate their *Super Pi* scores. For the VNS, given that the paper was published in 2010, we decided to use the same scaling factor as the one used for the GVNS (0.32), because of the proximity of their publication dates. As for ELNS Schmid and Ehmke (2017), the authors did not report comparisons based on CPU times, and we were not able to estimate their *Super Pi* score by only relying on the family of their CPU and its clock rate. Therefore, we assumed similar performance to our setup. In the remainder of this section, the time values reported in previous works are adjusted by the associated factors, in order to account for CPU differences. For example, the adjusted computational time of a solution obtained by I3CH can be obtained by multiplying its CPU time by 0.95.

Tables 3 to 5 summarize the results achieved by each algorithm. Comparison of solution quality is usually drawn in terms of relative percentage error (*rpe*) with respect to the best-known solution (BKS) for the

Algorithm	CPU	Super Pi Estimate	Factor
VNS	2.4 GHz CPU (reference unknown)	Unknown	≤ 0.32
GRASP-ELS	Intel Pentium 4 processor, 3.00 GHz	44.3	0.32
GVNS	Intel Pentium (R) IV, 3 GHz CPU	44.3	0.32
I3CH	Intel Xeon E5430 CPU clocked at 2.66 GHz	14.7	0.95
ELNS	Intel Xeon 3.1 GHz (reference unknown)	Unknown	≈ 1
MS-ILS	Intel Xeon X7542 CPU at 2.67GHz	14.1	1

Table 2: Estimation of single-thread performance.

standard benchmark, and in terms of average relative percentage error (*arpe*) for the "OPT" benchmark, but we chose to include comparisons on the basis of *rpe* and *arpe* for both benchmarks. These two metrics are computed as: $rpe = \frac{(BKS - Z_{max})}{BKS} * 100\%$ and $arpe = \frac{(BKS - Z_{avg})}{BKS} * 100\%$, where Z_{max} denotes the best score obtained over different runs and Z_{avg} the average score. Column cpu_{avg} of each table reports the average computational time of the above mentioned algorithms in seconds. The detailed results obtained by our MS-ILS are presented in the supplementary material associated with this paper, where they are compared to the best-known solutions in the literature. These results are presented in tables, of which, each consists of two identically structured parts. Each part contains the name of the instance, the best-known solution (BKS) to the instance, including the ones found by our method and the optimal solutions of Duque et al. (2015) and Tae and Kim (2015), the maximum score (Z_{max}) obtained by our algorithm, the relative error (*rpe*), the average score (Z_{avg}), the average error (*arpe*), and the average computational time in seconds (cpu_{avg}).

Table 3 reports the results obtained by all the state-of-the-art algorithms and the MS-ILS using the previous configuration on the standard benchmark. As can be seen in the table, our MS-ILS achieves an average relative gap lower than those achieved by the other algorithms for every set of instances. As for the relative gap, it achieves better results than the literature on most instances, apart from instances $rc200$ with $m = 3$, and $rc100$ with $m = 4$ where ELNS does slightly better, but at the cost of higher computational effort. On Cordeau's instances, our method requires a little more computation time to find high-quality solutions compared to the likes of the GRASP-ELS and the GVNS, but is still much faster than I3CH and the VNS, and the quality of the solutions it obtains is much better than that of the other algorithms. Furthermore, note that Solomon's instances with wider time windows become easier to solve as m becomes larger.

The performance of our algorithm on the "OPT" data set is shown in Tables 4 and 5. The column $\#OPT$ indicates the number of optimal solutions found by the MS-ILS. Note that, in the case of column $\#OPT$, the value reported at the bottom of the two tables corresponds to the sum of the values of the column. Note also that the authors of GRASP-ELS (Labadie et al., 2011) and GVNS (Labadie et al., 2012) only report the average value of solutions obtained over several runs. Most of the published metaheuristics reported their results using the average relative gap *arpe*, except for Hu and Lim (2014) who used only the best relative gap *rpe*. For this reason, we split the comparisons into two: Table 4 displays comparisons based on the *arpe*,

Instances	VNS			GRASP-ELS			GVNS			I3H			ELNS			MS-ILS		
	rpe%	*arpe*%	cpu	*rpe*%	*arpe*%	cpu	*rpe*%	*arpe*%	cpu	*rpe*%	*arpe*%	cpu	*rpe*%	*arpe*%	cpu	*rpe*%	*arpe*%	cpu
m=1																		
c100	**0.00**	0.11	31.5	**0.00**	**0.00**	7.2	0.56	1.22	53.3	0.00	-	24.0	**0.00**	**0.00**	19.0	**0.00**	**0.00**	2.0
r100	**0.00**	0.05	28.5	0.11	0.22	1.1	1.72	2.68	9.4	0.56	-	27.2	**0.00**	0.06	15.2	**0.00**	**0.02**	2.1
rc100	**0.00**	0.04	20.9	0.33	0.40	0.6	1.88	3.51	3.1	1.66	-	24.3	**0.00**	0.31	10.5	**0.00**	**0.02**	1.1
c200	**0.00**	0.21	179.3	0.40	0.61	10.3	0.55	1.11	61.6	0.40	-	80.2	**0.00**	**0.08**	47.0	**0.00**	**0.08**	9.9
r200	0.95	1.60	341.1	1.14	2.15	3.6	2.98	3.90	10.8	1.58	-	167.4	0.05	0.29	65.6	**0.03**	**0.10**	22.9
rc200	0.25	1.52	278.2	1.55	2.37	2.6	2.70	4.13	5.1	2.85	-	113.4	0.23	0.48	47.2	**0.00**	**0.12**	13.4
pr01-pr10	**0.02**	1.10	263.1	0.75	1.46	1.6	0.56	1.62	4.0	1.07	-	103.6	0.10	**0.18**	40.1	**0.02**	**0.18**	13.0
pr11-pr20	1.44	3.41	334.7	2.20	3.42	2.5	3.21	4.30	7.8	4.31	-	123.7	1.44	2.08	67.5	0.85	1.15	21.5
m=2																		
c100	**0.00**	0.27	28.2	**0.00**	0.07	22.7	0.47	0.72	44.7	**0.00**	-	24.0	0.17	0.20	30.6	**0.00**	**0.03**	3.6
r100	0.18	1.43	20.3	1.04	1.78	2.5	1.22	1.83	19.3	0.61	-	27.2	0.14	0.29	25.5	**0.00**	**0.04**	2.7
rc100	0.23	1.46	17.7	1.46	2.32	1.5	0.78	2.80	6.5	0.90	-	24.3	**0.00**	**0.05**	20.7	**0.00**	**0.05**	2.9
c200	0.59	0.95	174.6	0.17	0.34	9.4	0.34	0.66	10.8	0.76	-	80.2	**0.00**	0.20	63.5	**0.00**	**0.14**	16.9
r200	0.85	1.35	324.8	0.93	1.25	5.6	1.27	1.90	4.7	0.81	-	167.4	0.27	0.50	42.2	**0.08**	**0.26**	30.9
rc200	1.06	2.04	257.5	1.22	1.73	5.5	2.24	3.12	4.1	1.18	-	113.4	0.32	0.70	52.5	**0.15**	**0.39**	25.9
pr01-pr10	1.10	4.11	167.9	1.34	2.42	6.2	1.05	2.02	12.5	1.34	-	103.6	0.72	1.20	85.0	**0.10**	**0.43**	18.1
pr11-pr20	1.95	4.31	198.0	3.12	4.25	9.2	1.91	2.88	26.4	3.39	-	123.7	2.06	2.98	135.0	**0.84**	**1.29**	32.1
m=3																		
c100	0.11	0.73	27.4	0.24	0.56	27.8	0.45	0.95	52.8	0.11	-	82.6	0.11	0.49	40.5	**0.00**	**0.09**	5.1
r100	0.23	1.48	19.8	0.91	1.58	4.4	1.23	2.28	23.7	0.22	-	59.9	0.08	0.24	35.3	**0.00**	**0.06**	3.5
rc100	0.38	1.42	19.4	1.85	2.83	2.8	0.93	2.34	10.8	0.29	-	56.0	0.01	0.37	30.2	**0.00**	**0.25**	2.9
c200	0.30	1.01	63.0	0.58	0.92	8.6	0.77	1.29	17.7	0.16	-	381.2	0.22	0.42	27.7	**0.09**	**0.18**	5.2
r200	0.09	0.16	102.9	0.06	0.07	0.8	0.17	0.26	2.2	0.07	-	500.5	0.06	0.08	4.9	**0.03**	**0.06**	3.2
rc200	0.11	0.32	129.3	0.13	0.26	2.7	0.32	0.44	2.4	0.04	-	417.7	**0.06**	**0.11**	14.8	0.07	**0.11**	8.0
pr01-pr10	1.92	4.03	151.4	1.73	2.39	13.0	0.77	1.54	27.5	0.77	-	234.7	1.29	1.97	124.2	**0.34**	**0.74**	25.7
pr11-pr20	2.51	4.06	165.6	3.02	3.97	13.7	1.74	2.52	48.2	1.84	-	289.4	2.64	3.40	191.7	**0.48**	**0.94**	43.3
m=4																		
c100	0.38	1.34	26.2	0.79	1.12	27.1	1.14	1.72	42.6	0.20	-	180.7	0.77	1.26	48.5	**0.10**	**0.31**	7.6
r100	0.37	1.61	19.6	1.01	1.72	7.7	1.28	2.35	27.1	0.23	-	112.4	0.14	0.38	42.7	**0.10**	**0.18**	4.4
rc100	0.58	2.45	18.7	1.67	2.41	4.3	1.08	1.92	11.8	0.36	-	95.9	**0.10**	0.35	37.7	0.11	**0.33**	3.5
c200	0.00	0.00	33.5	0.00	0.00	0.0	0.00	0.00	0.2	0.00	-	158.2	0.00	0.00	0.0	**0.00**	**0.00**	0.1
r200	0.00	0.00	48.2	0.00	0.00	0.0	0.00	0.00	0.1	0.00	-	86.3	0.00	0.00	0.0	**0.00**	**0.00**	0.1
rc200	0.00	0.00	52.7	0.00	0.00	0.0	0.00	0.01	0.3	0.00	-	155.9	0.00	0.00	0.0	**0.00**	**0.00**	0.1
pr01-pr10	2.63	4.31	129.0	2.65	3.27	14.6	1.84	2.39	40.7	1.13	-	402.8	2.48	3.32	156.1	**0.42**	**0.85**	34.4
pr11-pr20	3.12	4.29	130.6	3.40	4.22	20.9	2.86	3.62	74.4	1.29	-	472.1	3.33	4.11	234.9	**0.48**	**1.08**	59.1
Average	0.67	1.60	118.9	1.06	1.57	7.5	1.19	1.94	20.8	0.88	-	156.7	0.52	0.82	54.9	**0.13**	**0.30**	13.3

Table 3: Comparison of the MS-ILS to the state-of-the-art methods on the standard benchmark.

Instances	Nb.	VNS			GRASP-ELS			GVNS			ELNS			MS-ILS		
		#OPT	*arpe*%	cpu	#OPT	*arpe*%	cpu	#OPT	*arpe*%	cpu	#OPT	*arpe*%	cpu	#OPT	*arpe*%	cpu
c100	9	**9**	0.02	6.4	-	**0.00**	0.4	-	0.47	2.5	**9**	**0.00**	1.5	**9**	**9**	0.5
r100	12	1	0.50	6.9	-	0.73	33.7	-	1.55	12.6	4	0.73	45.4	**5**	**0.47**	14.7
rc100	8	**4**	0.85	6.7	-	0.90	25.2	-	1.29	12.6	3	0.39	39.1	**4**	**0.37**	8.0
c200	8	-	-	-	-	0.00	0.0	-	0.00	0.2	**8**	**0.00**	0.0	**8**	**0.00**	0.1
r200	11	-	-	-	-	0.04	2.1	-	0.17	1.8	**11**	0.01	9.7	**11**	**0.00**	2.9
rc200	8	-	-	-	-	0.03	0.9	-	0.16	0.9	**8**	**0.00**	2.6	**8**	**0.00**	1.4
pr01-pr10	10	5	1.30	22.9	-	**0.92**	22.9	-	1.25	16.4	3	1.22	154.9	**5**	0.95	25.8
Avg./Total	66	19	0.67	10.7	-	0.37	12.2	-	0.70	6.7	46	0.34	36.2	**50**	**0.26**	7.6

Table 4: Performance comparison based on *arpe* average for "OPT" data set.

Instances	Nb.	VNS			I3H			ELNS			MS-ILS		
		#OPT	*rpe*%	cpu	#OPT	*rpe*%	cpu	#OPT	*rpe*%	cpu	#OPT	*rpe*%	cpu
c100	9	**9**	**0.00**	6.4	**9**	**0.00**	45.2	**9**	**0.00**	1.5	**9**	**0.00**	0.5
r100	12	1	0.20	6.9	**8**	**0.07**	833.8	4	0.58	45.4	5	0.34	14.7
rc100	8	4	0.37	6.7	**8**	**0.00**	54.5	3	0.29	39.1	4	0.19	9.4
c200	8	-	-	-	**8**	**0.00**	0.6	**8**	**0.00**	0.0	**8**	**0.00**	0.1
r200	11	-	-	-	9	0.07	164.7	**11**	**0.00**	9.7	**11**	**0.00**	2.9
rc200	8	-	-	-	7	0.04	180.7	**8**	**0.00**	2.6	**8**	**0.00**	1.4
pr01-pr10	10	5	1.06	22.9	**6**	**0.78**	310.3	3	1.06	154.9	5	0.86	26.4
Avg./Total	66	19	0.41	10.7	**55**	**0.14**	227.1	46	0.27	36.2	50	0.20	7.9

Table 5: Performance comparison based on *rpe* average for "OPT" data set.

while Table 5 shows the comparisons based on the *rpe*. Even though the "OPT" data set is known to be difficult to solve, MS-ILS achieves both the smallest average relative gap and the second smallest relative gap, and is able to obtain 59 out of 66 optimal solutions in a reasonable amount of time. The I3CH is the only other algorithm that finds slightly better solutions than the MS-ILS, but it requires significantly higher computational times to do so. In some cases, it is up to twenty times slower than the MS-ILS.

For a more thorough comparison, we conducted additional experiments using other combinations of values for parameters $iter_{max}$, $iter_{ils}$ and $Size_{\mathcal{M}}$ in order to achieve smaller and larger computation times than those presented above. We then compared the performance of every combination with those of the state-of-the-art algorithms, namely VNS, GRASP-ELS, GVNS, I3CH and ELNS in terms of computation time (CPU) and in terms of either relative error(*rpe*) or average relative error (*arpe*). The combinations of values used for these experiments were chosen based on the results of the Pareto test described in Section 5.2.

Figures 10 and 9 show the performance of the MS-ILS using different combinations of parameters compared to state-of-the-art algorithms; the first in terms of computation time and relative error (*rpe*), the second in terms of *cpu* and average relative error (*arpe*). As can be seen in both figures, the MS-ILS achieves better results compared to the remaining algorithms: not only does it achieve smaller deviations when computation times are equivalent, but each combination of parameter values we tested resulted in smaller deviations

compared to the remaining algorithms.

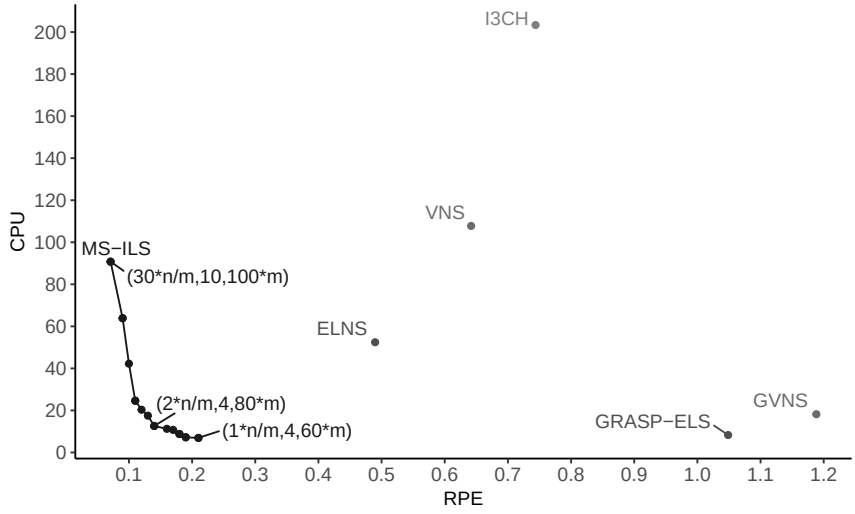


Figure 9: Comparison of different MS-ILS settings with the literature based on *cpu* and *rpe*.

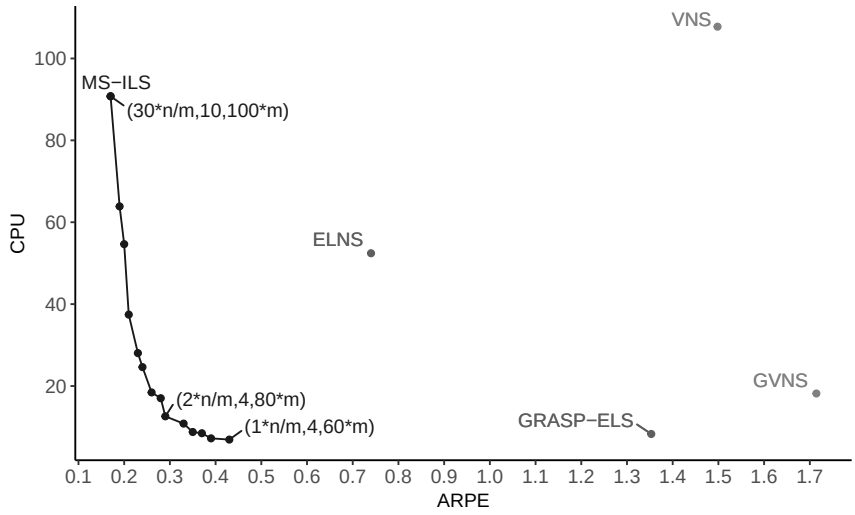


Figure 10: Comparison of different MS-ILS settings with the literature based on *cpu* and *arpe*.

Tables 6 and 7 display the results of two runs of the MS-ILS using two different settings of $iter_{max}$, $iter_{ils}$ and $Size_{\mathcal{M}}$. The first setting is the same as the one used in the above comparisons with the literature. The second one is a slower version of the algorithm obtained by setting $iter_{max} = 30n/m$, $iter_{ils} = 10$ and $Size_{\mathcal{M}} = 100m$. The main observation here is that, if allowed to run longer, the MS-ILS is able to further improve solution quality. It is able to find the best known solutions for all of Solomon's instances at least one time out of five. On the "OPT" benchmark, it can obtain the optimal solutions for 59 out of 66 instances. Overall, it is able to find the best known solutions for 93% of the instances available in the literature.

During our experiments, including the tuning experiments, the MS-ILS found 57 new best-known solution

Instances	MS-ILS (2,4,80)			MS-ILS (30,10,100)		
	$rpe\%$	$arpe\%$	cpu	$rpe\%$	$arpe\%$	cpu
m=1						
c100	0.00	0.00	2.0	0.00	0.00	32.5
r100	0.00	0.02	2.1	0.00	0.00	25.9
rc100	0.00	0.02	1.1	0.00	0.00	20.0
c200	0.00	0.08	9.9	0.00	0.00	80.4
r200	0.03	0.10	22.9	0.00	0.03	167.1
rc200	0.00	0.12	13.4	0.00	0.05	131.8
pr01-pr10	0.02	0.18	13.0	0.00	0.01	131.2
pr11-pr20	0.85	1.15	21.5	0.85	0.97	205.0
m=2						
c100	0.00	0.03	3.6	0.00	0.00	30.2
r100	0.00	0.04	2.7	0.00	0.01	23.8
rc100	0.00	0.05	2.9	0.00	0.01	20.8
c200	0.00	0.14	16.9	0.00	0.07	85.0
r200	0.08	0.26	30.9	0.01	0.19	187.9
rc200	0.15	0.39	25.9	0.02	0.22	171.9
pr01-pr10	0.10	0.43	18.1	0.03	0.20	124.4
pr11-pr20	0.84	1.29	32.1	0.69	0.94	254.1
m=3						
c100	0.00	0.09	5.1	0.00	0.04	39.8
r100	0.00	0.06	3.5	0.00	0.00	27.5
rc100	0.00	0.25	2.9	0.00	0.09	26.1
c200	0.09	0.18	5.2	0.03	0.11	32.1
r200	0.03	0.06	3.2	0.00	0.05	18.7
rc200	0.07	0.11	8.0	0.00	0.09	36.5
pr01-pr10	0.34	0.74	25.7	0.12	0.41	168.7
pr11-pr20	0.48	0.94	43.3	0.12	0.64	349.4
m=4						
c100	0.10	0.31	7.6	0.10	0.22	44.7
r100	0.10	0.18	4.4	0.01	0.09	34.9
rc100	0.11	0.33	3.5	0.01	0.08	31.8
c200	0.00	0.00	0.1	0.00	0.00	0.1
r200	0.00	0.00	0.1	0.00	0.00	0.1
rc200	0.00	0.00	0.1	0.00	0.00	0.1
pr01-pr10	0.42	0.85	34.4	0.07	0.58	241.2
pr11-pr20	0.48	1.08	59.1	0.17	0.73	415.7
Average	0.13	0.30	13.3	0.07	0.18	98.7

Table 6: Comparison of two different settings of MS-ILS on the standard benchmark.

Instances	Nb.	MS-ILS (2,4,80)				MS-ILS (30,10,100)			
		#OPT	rpe%	$arpe$%	cpu	#OPT	rpe%	$arpe$%	cpu
c100	9	9	0.00	0.00	0.5	9	0.00	0.00	0.6
r100	12	5	0.34	0.47	14.7	10	0.02	0.09	92.7
rc100	8	4	0.19	0.37	9.4	7	0.08	0.19	58.5
c200	8	8	0.00	0.00	0.1	8	0.00	0.00	0.1
r200	11	11	0.00	0.00	2.9	11	0.00	0.00	1.4
rc200	8	8	0.00	0.00	1.4	8	0.00	0.00	1.8
pr01-pr10	10	5	0.86	0.95	26.4	6	0.64	0.71	144.8
Avg./Total	66	50	0.20	0.26	7.9	59	0.10	0.14	42.8

Table 7: Comparison of two different settings of MS-ILS on the "OPT".

values for the standard benchmark instances: 20 for Solomon's instances and 37 for Cordeau's. Table 8 reports the new best-known solution values found by MS-ILS. For each instance, it indicates the number of vehicles and the new solution value.

Instance	m	Old BKS	New BKS	Instance	m	Old BKS	New BKS	Instance	m	Old BKS	New BKS
r207	1	1077	1078	pr04	2	926	928	pr02	4	1079	1083
r208	1	1117	1118	pr05	2	1101	1103	pr03	4	1232	1248
r209	1	961	962	pr10	2	1134	1145	pr04	4	1585	1595
r210	1	1000	1002	pr13	2	843	845	pr05	4	1838	1859
rc204	1	1140	1143	pr15	2	1220	1238	pr06	4	1860	1898
rc208	1	1057	1058	pr18	2	953	955	pr08	4	1382	1392
				pr19	2	1034	1041	pr09	4	1619	1626
r201	2	1256	1260	pr20	2	1237	1251	pr10	4	1943	1965
r202	2	1350	1353					pr12	4	1132	1135
r203	2	1420	1431	pr02	3	943	947	pr13	4	1386	1392
r205	2	1395	1402	pr03	3	1010	1014	pr14	4	1670	1688
r206	2	1447	1452	pr04	3	1294	1298	pr15	4	2065	2085
r209	2	1419	1423	pr05	3	1482	1500	pr17	4	934	936
r210	2	1430	1438	pr06	3	1514	1519	pr18	4	1539	1558
rc201	2	1385	1386	pr08	3	1139	1142	pr19	4	1750	1780
rc202	2	1520	1523	pr10	3	1573	1582	pr20	4	2062	2115
rc203	2	1637	1640	pr13	3	1145	1159				
rc204	2	1716	1718	pr14	3	1372	1375				
rc207	2	1601	1609	pr15	3	1654	1694				
rc208	2	1691	1705	pr18	3	1281	1289				
				pr19	3	1417	1428				
r201	3	1441	1450	pr20	3	1684	1722				

Table 8: New best-known solutions values found by MS-ILS.

Altogether, our MS-ILS is very competitive. It is able to find the current best-known solutions for 78% of the literature instances and was able to improve the best-known solutions for many of them. It achieves an average relative error ($arpe$) of only 0.30% on the standard benchmark and 0.26% on the "OPT" benchmark, and a relative percentage error (rpe) of 0.13% and 0.20%, on the two benchmarks, respectively, while still being faster than most state-of-the-art algorithms. In comparison, the previous best performing approach in

the literature finds 65% of the current best-known solutions, and achieves an *arpe* of 0.80% on the standard benchmark and 0.34% on the "OPT" benchmark, and a *rpe* of 0.51% and 0.27%, on the two benchmarks, respectively. Finally, our experiments show that the MS-ILS can be tuned to either favor speed or solution quality and remain relatively better than other approaches.

5.4. Performance analysis

In the following, we discuss the results of the experiments conducted to evaluate the impact of each of the key components of our algorithm, namely the combination of two different search spaces and the adaptive memory. To do this, we derive alternative versions of the MS-ILS by disabling search spaces one at a time, or by disabling the adaptive memory. The considered configurations are as follows:

a. **Standard**: the standard algorithm described in Section 4.

b. **No Routes**: an alternate version of MS-ILS where the route search space operators are disabled.

c. **No Tours**: an alternate version of MS-ILS where the giant tour search space operators and the splitting procedures are disabled. To construct solutions from the memory, we simply select m routes that do not share customers.

d. **No Memory**: version of the standard algorithm where the adaptive memory and giant tour construction are disabled. Each iteration of the main loop starts from a randomly generated solution.

All of the above configurations were run ten times on each instance of both the standard and the "OPT" benchmark using the parameter values given in Section 5.2. Table 9 shows the average gap relative to the best know solutions, and the average CPU time achieved by each of the configurations from (a) to (d).

Comparisons between the configurations (a) to (c) highlight the impact of alternating between route and giant tour search spaces instead of only using one of them. Disabling either one of the search spaces translates into a decrease of solution quality compared to configuration (a), but removing the route operators has a bigger impact on solution quality than removing giant tour operators. We still decided to keep both spaces in the design of the algorithm because both of them help improve solution quality; besides, the giant tour search spaces is still needed to find good solutions for some instances.

Comparisons between configurations (a) and (d) show the contribution of including the adaptive memory into the multi-start framework. As we can see, adding the adaptive memory significantly improves the average gap on the two benchmarks, with a reasonable impact on CPU time. In terms of solution quality, the use of the adaptive memory has a more significant impact when solving instances of the standard benchmark with $m \geq 2$ than when solving instances with $m = 1$. This is because when $m = 1$, the algorithm chooses a previous local optimum and tries to improve it, whereas when $m \geq 2$, the algorithm constructs new solutions using several from previous local optima.

Most of the time, disabling one component results in a decrease in computation times. However, depending on the instance, the opposite can happen, and disabling one component may result in larger computation

		(a) Standard		(b) No Routes		(c) No Tours		(d) No Memory	
	Instances	arpe	cpu	arpe	cpu	arpe	cpu	arpe	cpu
m=1	Solomon 100	0.02 %	2.3	0.92 %	1.0	0.06 %	2.9	0.07 %	3.1
	Solomon 200	0.09 %	21.1	1.55 %	8.1	0.15 %	18.0	0.45 %	8.6
	Cordeau	0.57 %	21.8	4.50 %	6.8	0.69 %	19.5	1.03 %	21.6
	avg	0.23 %	15.1	2.32 %	5.3	0.30 %	13.5	0.52 %	11.1
m=2	Solomon 100	0.03 %	3.7	1.17 %	1.6	0.04 %	3.4	0.37 %	2.4
	Solomon 200	0.31 %	26.8	1.65 %	46.0	0.42 %	12.6	0.98 %	6.5
	Cordeau	0.83 %	31.5	4.98 %	8.8	1.05 %	21.0	2.66 %	14.9
	avg	0.39 %	20.7	2.60 %	18.8	0.50 %	12.3	1.33 %	8.0
m=3	Solomon 100	0.12 %	4.5	1.30 %	6.5	0.12 %	3.7	0.85 %	2.3
	Solomon 200	0.28 %	6.6	1.57 %	4.4	0.32 %	4.7	1.39 %	1.9
	Cordeau	0.83 %	42.3	5.03 %	13.7	1.00 %	24.8	3.37 %	13.2
	avg	0.41 %	17.8	2.63 %	8.2	0.48 %	11.1	1.87 %	5.8
m=4	Solomon 100	0.28 %	6.7	1.72 %	4.3	0.33 %	4.7	1.49 %	2.0
	Solomon 200	0.00 %	0.1	0.00 %	0.1	0.00 %	0.1	0.00 %	0.1
	Cordeau	0.84 %	54.2	5.44 %	22.2	1.07 %	24.8	3.97 %	12.8
	avg	0.37 %	20.3	2.39 %	8.9	0.47 %	9.9	1.82 %	5.0
OPT	Solomon 100	0.31 %	11.2	0.97 %	10.3	0.55 %	2.5	0.70 %	2.3
	Solomon 200	0.00 %	1.7	0.10 %	16.7	0.00 %	0.9	0.02 %	1.1
	Cordeau	0.99 %	28.2	1.76 %	70.3	1.10 %	6.7	1.42 %	10.3
	avg	0.44 %	13.7	0.94 %	32.5	0.55 %	3.3	0.72 %	4.6

Table 9: Performance analysis of MS-ILS components.

times. For example, in the standard benchmark on Solomon's instances with $m = 1$; disabling the route space local search operators hinders the progress of the algorithm towards good solutions, which explains the increase in CPU time.

6. Conclusion

In this paper, we introduced a very effective Multi-Start Iterated Local Search (MS-ILS) for the Team Orienteering Problem with Time Windows. Our algorithm is based on a local search that alternates between two different search spaces: the *route search space* that corresponds to actual solutions to the TOPTW, and the *giant tour search space* that makes it possible to explore the solution space without being limited by time windows and length constraints. In order to improve solution quality, several local search operators are included to be used in each search space. Additionally, the algorithm integrates an adaptive memory mechanism to further improve performance by making use of previous local optima to build better solutions.

Computational results have shown that our approach performs very well compared to state-of-the-art

algorithms and is able to outperform them in terms of overall solution quality and computation times. In particular, the MS-ILS is able to find the current best-known solutions, or better ones, for 78% of the benchmark instances within reasonable runtimes, and achieves an overall average relative gap of 0.30% and 0.26% on the two benchmarks of the literature, respectively. If given more time, the MS-ILS finds the current best-known solutions for 89% of the literature instances. Furthermore, our approach was also able to find new best solutions for 57 instances for which no optimal solution has yet been found.

Finally, the method proposed herein is flexible in the sense that it can be easily adapted to accommodate new constraints or to address other variants of the problem. One potential direction for future research would be to extend our algorithm to more realistic versions of orienteering problems with time windows.

Acknowledgments. The authors would like to thank the reviewers for their valuable comments and suggestions which improved the quality of the paper. This work is supported by the French Environment and Energy Management Agency (ADEME), the Hauts-de-France region and the European Regional Development Fund (ERDF). It was carried out within the framework of the Labex MS2T, funded by the French Government (Reference ANR-11-IDEX-0004-02). It is also partially supported by the TCDU project (Collaborative Transportation in Urban Distribution, ANR-14-CE22-0017).

References

Bouly, H., Dang, D.C., Moukrim, A., 2010. A memetic algorithm for the team orienteering problem. 4OR 8, 49–70.

Cabrera, N., Medaglia, A.L., Lozano, L., Duque, D., 2020. An exact bidirectional pulse algorithm for the constrained shortest path. Networks. In press.

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

Cura, T., 2014. An artificial bee colony algorithm approach for the team orienteering problem with time windows. Computers & Industrial Engineering 74, 270 – 290.

Dang, D.C., Guibadj, R.N., Moukrim, A., 2013. An effective pso-inspired algorithm for the team orienteering problem. European Journal of Operational Research 229, 332–344.

Duque, D., Lozano, L., Medaglia, A.L., 2015. Solving the orienteering problem with time windows via the pulse framework. Computers & Operations Research 54, 168 – 176.

Gedik, R., Kirac, E., Milburn, A.B., Rainwater, C., 2017. A constraint programming approach for the team orienteering problem with time windows. Computers & Industrial Engineering 107, 178 – 195.

Gendreau, M., Laporte, G., Semet, F., 1998. A tabu search heuristic for the undirected selective travelling salesman problem. European Journal of Operational Research 106, 539 – 545.

Golden, B.L., Levy, L., Vohra, R., 1987. The orienteering problem. Naval Research Logistics (NRL) 34, 307–318.

Guibadj, R.N., Moukrim, A., 2014. Memetic algorithm with an efficient split procedure for the team orienteering problem with time windows, in: International Conference on Artificial Evolution. EA 2013. Lecture Notes in Computer Science, vol 8752, Springer International Publishing, Cham. pp. 183–194.

Gunawan, A., Lau, H.C., Lu, K., 2015a. An iterated local search algorithm for solving the orienteering problem with time windows, in: Evolutionary Computation in Combinatorial Optimization, Springer International Publishing, Cham. pp. 61–73.

Gunawan, A., Lau, H.C., Lu, K., 2015b. Sails: hybrid algorithm for the team orienteering problem with time windows, in: Proceedings of the 10th international conference of the practice and theory of automated timetabling (patat 2014), MISTA, York, United Kingdom. pp. 202–217.

Gunawan, A., Lau, H.C., Lu, K., 2015c. Well-tuned ils for extended team orienteering problem with time windows, in: LARC Technical Report Series. Singapore Management University.

Gunawan, A., Lau, H.C., Vansteenwegen, P., 2016. Orienteering problem: A survey of recent variants, solution approaches and applications. European Journal of Operational Research 255, 315 – 332.

Hu, Q., Lim, A., 2014. An iterative three-component heuristic for the team orienteering problem with time windows. European Journal of Operational Research 232, 276–286.

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

Labadie, N., Melechovský, J., Wolfler-Calvo, R., 2011. Hybridized evolutionary local search algorithm for the team orienteering problem with time windows. Journal of Heuristics 17, 729–753.

Lin, S.W., Yu, V.F., 2012. A simulated annealing heuristic for the team orienteering problem with time windows. European Journal of Operational Research 217, 94–107.

Lozano, L., Medaglia, A.L., 2013. On an exact method for the constrained shortest path problem. Computers & Operations Research 40, 378 – 384.

Mendoza, J.E., Villegas, J.G., 2013. A multi-space sampling heuristic for the vehicle routing problem with stochastic demands. Optimization Letters 7, 1503–1516.

Montemanni, R., Gambardella, L.M., 2009. Ant colony system for team orienteering problems with time windows. Foundations of Computing and Decision Sciences 34, 287–306.

Montemanni, R., Weyland, D., Gambardella, L.M., 2011. An enhanced ant colony system for the team orienteering problem with time windows, in: 2011 International Symposium on Computer Science and Society, pp. 381–384.

Montoya, A., Guéret, C., Mendoza, J.E., Villegas, J.G., 2016. A multi-space sampling heuristic for the green vehicle routing problem. Transportation Research Part C: Emerging Technologies 70, 113 – 128.

Prins, C., 2004. A simple and effective evolutionary algorithm for the vehicle routing problem. Computers & Operations Research 31, 1985–2002.

Prins, C., Labadie, N., Reghioui, M., 2009. Tour splitting algorithms for vehicle routing problems. International Journal of Production Research 47, 507–535.

Righini, G., Salani, M., 2008. New dynamic programming algorithms for the resource constrained elementary shortest path problem. Networks 51, 155–170.

Righini, G., Salani, M., 2009. Decremental state space relaxation strategies and initialization heuristics for solving the orienteering problem with time windows with dynamic programming. Computers & Operations Research 36, 1191–1203.

Rochat, Y., Taillard, É.D., 1995. Probabilistic diversification and intensification in local search for vehicle routing. Journal of Heuristics 1, 147–167.

Sadykov, R., Vanderbeck, F., 2013. Bin packing with conflicts: A generic branch-and-price algorithm. INFORMS Journal on Computing 25, 244–255.

Schmid, V., Ehmke, J.F., 2017. An effective large neighborhood search for the team orienteering problem with time windows, in: Computational Logistics, Springer International Publishing, Cham. pp. 3–18.

Solomon, M.M., 1987. Algorithms for the vehicle routing and scheduling problems with time window constraints. Operations Research 35, 254–265.

Souffriau, W., Vansteenwegen, P., Vanden Berghe, G., Van Oudheusden, D., 2013. The multiconstraint team orienteering problem with multiple time windows. Transportation Science 47, 53–63.

Tae, H., Kim, B.I., 2015. A branch-and-price approach for the team orienteering problem with time windows. International Journal of Industrial Engineering 22, 243 – 251.

Takeo, Y., Seiji, K., Kohtaro, W., 2002. Heuristic and exact algorithms for the disjunctively constrained knapsack problem. Information Processing Society of Japan Journal 43, 2864–2870.

Tang, H., Miller-Hooks, E., 2005. A tabu search heuristic for the team orienteering problem. Computers & Operations Research 32, 1379–1407.

Tarjan, R.E., 1975. Graph theory and Gaussian elimination. Technical Report. Stanford University.

Tricoire, F., Romauch, M., Doerner, K.F., Hartl, R.F., 2010. Heuristics for the multi-period orienteering problem with multiple time windows. Computers & Operations Research 37, 351–367.

Vansteenwegen, P., Souffriau, W., Berghe, G.V., Oudheusden, D.V., 2009. Iterated local search for the team orienteering problem with time windows. Computers & Operations Research 36, 3281–3290.

Vansteenwegen, P., Souffriau, W., Oudheusden, D.V., 2011. The orienteering problem: A survey. European Journal of Operational Research 209, 1–10.