# An Effective Large Neighborhood Search for the Team Orienteering Problem with Time Windows

Verena Schmid[1] and Jan Fabian Ehmke[2]

[1] CD Laboratory for Efficient Intermodal Transport Operations, Department of Business Administration, University of Vienna, Vienna, Austria
[2] Management Science Group, Otto-von-Guericke University Magdeburg, Magdeburg, Germany, jan.ehmke@ovgu.de

**Abstract.** We propose an effective metaheuristic for the Team Orienteering Problem with Time Windows. The metaheuristic is based on the principle of Large Neighborhood Search and can outperform the performance of algorithms available in the literature. We provide computational experiments for well known benchmark instances and are able to compute new best solutions for 17 of these instances. On average, the gap between our results and best known solutions so far is below 1%, and our solution approach yields 70% of the best known solutions available in the literature. The new results can serve as benchmarks for future computational studies.

## 1 Introduction

Traditional routing problems such as the vehicle routing problem (VRP) or the vehicle routing problem with time windows (VRPTW) aim at cost-efficient service to a given number of customers. They consider a fixed fleet of vehicles and minimize total costs while guaranteeing feasibility of the resulting routes with respect to side constraints such as customer time windows, vehicle capacities, and total route durations. In this paper, we tackle a related problem: the Team Orienteering Problem with Time Windows (TOPTW). The TOPTW is a generalization of the well studied VRPTW, which tries to service a *subset* of potential customers. As opposed to the VRPTW, the complexity of the problem is extended by an additional degree of freedom, namely the choice to service a customer (or not). Every customer is associated with a profit, which may be collected upon visiting her. The goal is to maximize the total collected profit as opposed to minimizing total costs in the traditional VRP or VRPTW.

Possible applications of the TOPTW include, but are not limited to home fuel delivery [5], athlete recruiting from high schools [1], and the sport game of orienteering [2]. [22] consider the case of routing technicians to service customers in geographically distributed locations. Leisure related applications are discussed in [24], who present a personalized mobile tour guide for tourists in need for finding a plan to visit the most interesting sights. In [19] and [18], two approaches for selecting bars to be visited during a bar crawl are discussed.


T. Bektaş et al. (Eds.), ICCL 2017, LNCS 10572, pp. 3–18, 2017.
https://doi.org/10.1007/978-3-319-68496-3_1

Since orienteering problems belong to the class of NP-hard problems, it is unlikely that proven optimal solutions for the TOPTW can be found within polynomial time. Even when neglecting the quest for optimality, finding high-quality solutions within a reasonable amount of time remains a challenging task. For this reason, heuristics or more sophisticated metaheuristics seem to be a feasible way of tackling this problem. As pointed out in [4], despite the apparent simplicity of the orienteering problems, it is rather difficult to devise consistently good heuristics for these types of problems. This is partly due to the fact that profits and their locations and distances between locations are independent, and a good solution with respect to one criterion is often unsatisfactory with respect to the other. Hence, it is usually challenging to select the proper nodes albeit its feasible sequence that should be part of a (near-)optimal solution.

The sheer simplicity and the embedded computational complexity has attracted many researchers to investigate orienteering problems. [8] provide an excellent overview on the literature about orienteering problems and its applications. They give a formal description of the OP and present several relevant variants thereof. Within their survey, they extensively discuss and compare published exact and (meta)heuristic approaches presented so far. According to [8], the algorithms developed by [7] and [6] provide the largest proportion of current best known solutions so far. [7] present an iterated local search approach, which starts from an initial solution built with a greedy construction heuristic. The initial solution is improved by well-known local search components such as 2-OPT, SWAP and MOVE. They are able to improve a significant number of best known solutions from standard instances. [6] embed the iterated local search into a simulated annealing framework, which helps overcoming local optima.

Given related work, our aim is to provide a rather simple framework that solves the TOPTW effectively. We build our framework on an LNS-based metaheuristic. We embed the concept of forward time slack in the evaluation of adding or removing nodes from solutions in a smart way. We also investigate the pairwise removal of nodes, which turns out to be very effective for certain problem instances. Overall, by keeping the set of operators clear and manageable, we avoid the algorithm to be tuned and tailored to a specific set of instances. The contributions of this paper can be summarized as follows: i) We present an effective metaheuristic for solving the TOPTW, ii) we present innovative ways of choosing nodes to be added into any given solution, and iii) our approach is applied to a wide range of different types of instances for which the proposed algorithm performs exceptionally well and outperforms algorithms available in the literature.

The paper is structured as follows. We provide a mathematical formulation of the problem (Sect. 2) and describe the proposed solution approach in detail (Sect. 3). The performance of the algorithm is demonstrated on various sets of instances available in the literature (Sect. 4). We also provide a sensitivity analysis for the chosen parameter setting and compare our obtained results against the best ones available in the literature. We then conclude the main findings of this paper in Sect. 5.

## 2 Mathematical Problem Formulation

To formulate the TOPTW mathematically, we introduce the following notation: We consider a total number of $n$ (potential) customers, where $\mathcal{C} = \{1, \ldots, n\}$ denotes the set of customers. The fleet of $m$ homogeneous vehicles is referred to as set $\mathcal{K}$. Vehicles may start their routes from a central depot, which we will refer to as node 0. For modeling purposes, we also define an identical copy of that node as $n+1$. Hence, the set of all nodes is denoted as $\mathcal{V}$, where $\mathcal{V} = \mathcal{C} \cup \{0, n+1\}$. A time window $[e_i, a_i]$ is associated with every node $i \in \mathcal{V}$. Upon visiting node $i \in \mathcal{V}$, a profit of $p_i$ may be collected and it takes $d_i$ time units to do so. For depot nodes (i.e. for $i \in \{0, n+1\}$) $p_i = d_i = 0$. The time required to travel to node $j$ after $i$ is referred to as $t_{ij}$ $(\forall i, j \in \mathcal{V})$. The maximum route length is denoted as $T^{max}$. Let $M$ denote a sufficiently large number.

We introduce binary decision variables $y_i^k$ which evaluate to one if and only if node $i \in \mathcal{V}$ is visited by vehicle $k \in \mathcal{K}$. Additionally, we define binary decision variables $x_{ij}^k$, which will be equal to one if and only if vehicle $k \in \mathcal{K}$ attends node $j$ immediately after $i$, where $i, j \in \mathcal{V}$. Decision variables $s_i^k$ model the start of service of vehicle $k \in \mathcal{K}$ at node $i \in \mathcal{V}$. Then, the problem can be formulated as follows:

$$Z = \sum_{i \in \mathcal{V}} p_i \sum_{k \in \mathcal{K}} y_i^k \rightarrow \max \tag{1}$$

s.t.

$$\sum_{j \in \mathcal{V}} x_{ij}^k = y_i^k \qquad \forall i \in \mathcal{V} \backslash \{n+1\}, k \in \mathcal{K} \tag{2}$$

$$y_i^k = 1 \qquad \forall i \in \{0, n+1\}, k \in \mathcal{K} \tag{3}$$

$$\sum_{j \in \mathcal{V}} x_{ji}^k = y_i^k \qquad \forall i \in \mathcal{V} \backslash \{0\}, k \in \mathcal{K} \tag{4}$$

$$s_i^k + d_i + t_{ij} \leq s_j^k + M(1 - x_{ij}^k) \qquad \forall i, j \in \mathcal{V}, k \in \mathcal{K}, \text{ where } i \neq j \tag{5}$$

$$s_i^k \geq e_i y_i^k \qquad \forall i \in \mathcal{V}, k \in \mathcal{K} \tag{6}$$

$$s_i^k \leq a_i \qquad \forall i \in \mathcal{V}, k \in \mathcal{K} \tag{7}$$

$$\sum_{i \in \mathcal{V}} d_i y_i^k + \sum_{i \in \mathcal{V}} \sum_{j \in \mathcal{V}} t_{ij} x_{ij}^k \leq T^{max} \qquad \forall k \in \mathcal{K} \tag{8}$$

$$x_{ij}^k \in \{0, 1\} \qquad \forall i, j \in \mathcal{V}, k \in \mathcal{K} \tag{9}$$

$$y_i^k \in \{0, 1\} \qquad \forall i \in \mathcal{V}, k \in \mathcal{K} \tag{10}$$

$$s_i^k \geq 0 \qquad \forall i \in \mathcal{V}, k \in \mathcal{K}. \tag{11}$$

The objective function (1) maximizes the total collected profit. Constraints (2) and (4) ensure that nodes have a successor and predecessor along the route. Constraints (3) ensure that every route contains and consequently starts from and returns to the depot. Constraint (5) guarantees that the routes of all vehicles

are feasible and that sub tours are avoided. Due to Constraints (6) and (7), it is ensured that nodes may only be (started to be) serviced within the given time window. Constraints (8) ensure that the maximum duty time per vehicle is not exceeded. Finally, Constraints (9)–(11) restrict the feasible domain of the decision variables.

## 3 Solution Approach

As outlined above, the TOPTW can be seen as a generalization of a classical routing problem such as the VRPTW and hence is NP-hard. We propose to solve the problem with a metaheuristic based on Large Neighborhood Search (LNS), a metaheuristic which originally has been proposed by [15] for solving a pickup and delivery problem with time windows (PDPTW). Below, their ideas are extended, and additional problem specific operators are presented. LNS itself is an iterative metaheuristic which destroys and repairs a given solution consecutively. This concept has been proposed by [20], who describe the general idea of iteratively destroying (*ruin*) and repairing (*recreate*) solutions. Within the PDPTW and classical routing problems such as the VRPTW, all customers or requests need to be served. This is no longer the case for the TOPTW. Hence, specific operators are required to take care of the *choice* of customers to be visited.

Our algorithm works as follows. We generally employ the main ideas underlying the concept of LNS as proposed by [20] and [15]. First, an initial feasible solution $\mathcal{S}$ is generated. Then, the initial or, from the second iteration, current solution is destroyed and repaired. This is done until a given number of iterations $N$ has been reached. Solutions are compared based on their objective function value. Any solution improving (or tying with) the best solution obtained so far is stored in a pool of best solutions $\mathcal{S}^{best}$. The pool of best solutions is updated whenever a new improving solution has been found. Otherwise, a solution is chosen randomly from the set of best solutions found. To include some degree of diversity within the search process and avoid being stuck in a local optimum, after $R$ iterations without improving the current best solution, the current best solution is replaced through a randomly selected solution from the pool of best solutions. A technical outline of the proposed solution approach is depicted in Algorithm 1. Details on the operators are provided below.

The sketched components and procedures are described in more detail within the following subsections. In particular, we introduce several problem-specific operators to be used within this framework.

### 3.1 Solution Representation

To represent a solution within the algorithm, we encode the individual routes of all vehicles in use. To this end, we focus on the specific customers visited and their sequence within the route. The latter allows us to derive additional information with respect to the timing of visits, the waiting times that may occur in between, and any buffer time (slack) between any two customer nodes on a route that allows for rapid feasibility checks upon inserting new customers.

**Algorithm 1** A Large Neighborhood Search for the TOPTW

```
1: S ← GenerateInitialSequence                          ▷ generate initial solution
2: Z^best ← Z(S)                                        ▷ save objective of best solution so far
3: S^best ← {S}                                         ▷ initialize pool of best solutions
4: while termination criterion not reached do
5:     S' ← Destroy(S)                                  ▷ destroy solution
6:     S' ← Repair(S')                                  ▷ repair solution (& apply local search within)
7:     if Z(S') ≥ Z^best then                           ▷ (new) best solution found?
8:         Z^best ← Z(S')                               ▷ update objective of best solution so far
9:         update S^best                                ▷ update pool of best solutions
10:    else
11:        S ← any solution from S^best                 ▷ pick solution from pool
12:    end if
13: end while
```

### 3.2 Destroy Operators

In every iteration, we destroy the current solution by *removing* a number of customer nodes $n_D$ from the current set of routes, where $n_S$ denotes the total number of customers currently scheduled. Note that $n_D \leq n_S \leq n$. We remove up to $d\%$ of all customer nodes currently scheduled to be visited. The actual number of customer nodes to be selected for removal is chosen randomly from a discrete uniform distribution $n_D \sim U(1, d * n_S)$.

Traditionally, nodes are removed individually. This approach may lead to a suboptimal solution, which may be hard to improve. Imagine a route where a subsequence of nodes is far away from the remainder of the route (e.g. see nodes $i_s$ and $i_{s+1}$ in Figure 1a). Typically, the removal of a single node of the subsequence would neither lead to a significant reduction in travel time nor would there result a sufficient amount of slack upon removal for insertion of alternative nodes. Hence, we allow to remove *sequences of nodes*: rather than removing them individually, we remove several nodes simultaneously.

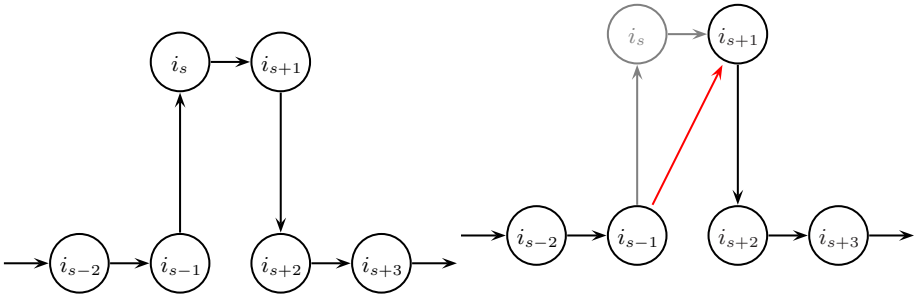
In particular, sequences of nodes are removed until the total number of nodes to be removed $n_D$ has been reached. The actual length evolves *iteratively*, i.e., the length of the sequence under consideration is extended gradually as long as the average savings in travel time increase. Note that, contrary to classical routing problems, not all customers need to be part of the solution for it to become feasible. Instead, only a subset of customers may be visited if beneficial for the objective function given that we are still able to satisfy all constraints.

More formally, we consider a route defined as a sequence of nodes $(0, i_1, i_2, \ldots, i_{n_k}, 0)$, where $n_k$ denotes the number of customers currently scheduled on route $k$.[3] The length $l$ of the sequence of nodes to be removed starting from node $i_s$ is extended as long as the following condition holds or the end of

[3] For improved readability we refrain from using an index referring to the actual route. The following considerations will be made independently for every route.

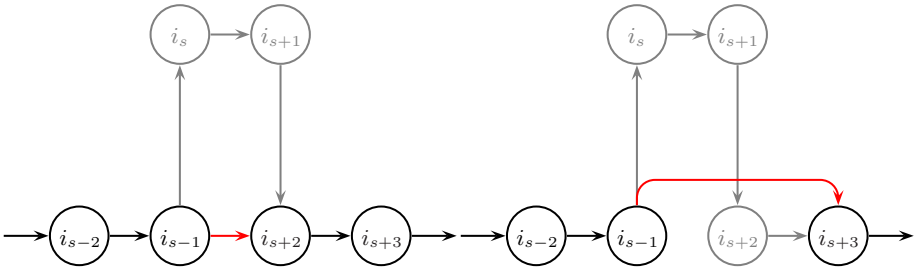
the route has been reached:

$$\frac{1}{l}(\sum_{p=s}^{s+l} t_{i_{p-1},i_p} - t_{i_{s-1},i_{s+l}}) < \frac{1}{l+1}(\sum_{p=s}^{s+l+1} t_{i_{p-1},i_p} - t_{i_{s-1},i_{s+l+1}}). \quad (12)$$




(a) original tour (before destroy)

(b) destroy operator starting at $i_s$ ($l = 1$)




(c) destroy operator starting at $i_s$ ($l = 2$)

(d) destroy operator starting at $i_s$ ($l = 3$)

Fig. 1: Estimating consequences for detouring nodes within the Destroy Operator

This approach has shown to be especially useful when the considered nodes are geographically clustered. The following Fig. 1 illustrates the underlying idea. Fig. 1a shows a subsequence of the original route before being destroyed. Fig. 1b-1d show the resulting changes if $l$ is set to 1, 2 and 3, respectively. Nodes and arcs that are about to be removed are shown in gray, new arcs to be added are highlighted in red. Assuming that the travel time along all horizontal (vertical) arcs in Fig. 1a equals 1 (2) and all travel times are to scale, the average savings are $2 - \sqrt{2}$ ($l = 1$), 1 ($l = 2$) and $0.\dot{6}$ ($l = 3$). As $1 > 0.\dot{6}$, the dynamic length $l$ would be set to 2, and the route would be destroyed as shown in Fig. 1c.

Once the number of nodes to be removed has been determined, a *destroy operator* is selected to identify consequences upon removal of particular nodes. We have developed the following five destroy operators, which focus on different important characteristics of a customer node with regard to solution quality:

- **Profit (P)** calculates the impact the removal of a node has upon the total profit as quantified by the objective function.
- **Travel Time (T)** calculates the impact the removal of a node has upon total travel time.
- **Potential (POT)** calculates the potential that the removal of a node yields for the insertion of another node. Details are given below.
- ***POT/TT*** denotes the potential of a removal of a node relative to the travel time reduction.
- ***POT*$^2$*/TT*** Alternative variant of ***POT/TT***, i.e., the squared potential relative to travel time.

Upon removal of nodes, slack time as defined by [17] may appear within a route. The slack time may result from arriving before the start of a time window and hence causes waiting time. Additionally, it may correspond to additional delays that could feasibly be considered, e.g. by postponing the start of a service within the time window without making the remainder of the route infeasible. The idea of operator **POT** is as follows: upon removal of a node, we investigate the possibility of inserting other nodes instead at the same position of the route. The potential is determined by the sum of the maximum profit of up to $o$ additional nodes to be inserted within the available slack time. Similar ideas are considered upon insertion of a node. Here, there might still be some additional slack left which is used to investigate the potential of $o$ additional nodes to be inserted thereafter.

Having quantified the impact of a node's removal on the solution quality, the particular nodes or node sequences are selected for removal. The removal of nodes is implemented according to one of the three following variants:

- **Random (R)** We randomly pick the node (sequence) from the set of candidates available for removal.
- **Greedy (G)** We delete the best node (sequence) according to the above destroy operators.
- **Bias (B)** We randomly pick the node (sequence), and the probabilities are defined according to the above measures of the destroy operators. For instance, when picking nodes based on their profit, the selection probabilities would reflect the relative proportions of the individual nodes' profits.

### 3.3 Repair Operators

Following a destroy operation, routes are reconstructed by a repair operator. In particular, in every iteration, customer nodes that are not part of the current solution are inserted back into the destroyed solution. To this end, we analyze the current routes in random order and compute the consequences of insertion

of each removed node at its best insertion position, which is derived according to smallest increase of travel time. From all possible nodes available for insertion, we select the one for insertion that fits best according to a particular repair operator. Following the ideas of the destroy operators, the repair operators evaluate the consequence of insertion of a customer node with regard to the quality of the current solution (P, T, POT, $POT/TT$, $POT^2/TT$). This procedure is finalized for a route once there are no more candidate nodes that could be inserted feasibly into the route, and we can continue with the next route. Note again that we can conduct the required feasibility checks quickly thanks to efficient slack computation as described by [17].

The generation of an initial solution for the TOPTW is trivial. We create an initial solution starting from an empty solution where no customer nodes are currently included. Then, we use the repair operators presented above to fill the initial solution.

## 4 Results

In this section, we investigate the performance of the proposed algorithm, compare it against benchmark results available in the literature, and justify the chosen parameter settings.

### 4.1 Benchmark Data and Computational Setup

We tested our algorithm on various sets of instances available in the literature. We present results for instances of both the TOPTW and OPTW (which is a special case of the TOPTW where $m = 1$). [16] propose two sets of instances for the TOPTW: The first set comprises 29 instances (which from now on will be referred to as *Solomon1* in general and c1*, r1*, rc1* in particular), which are based on the Solomon's instances for the VRPTW ([21]). Additionally, they propose 10 instances developed by [3] for the multi-depot vehicle routing problem (pr01-pr10). [13] were among the first to use a second test set of 27 instances from the Solomon set (*Solomon2*, c2*, r2*, rc2*). Finally, [14] proposed to use another set of 10 instances from the Cordeau data set (pr11-pr20). We well refer to pr01-10 and pr11-20 as *Cordeau1* and *Cordeau2*, respectively.

The number of customers corresponds to 100 for instances derived from the Solomon data set and is between 48 and 288 for instances derived from the Cordeau data set. Instances within *Solomon2* have wider time windows than those in *Solomon1*. As a consequence, the resulting routes tend to become longer. Distances between locations for Solomon instances were rounded down to the first decimal point and rounded down to the second decimal point for the instance set *Cordeau1* and *Cordeau2* as originally reported in [16].

For the TOPTW, [14] suggest to vary the number of vehicles in *Solomon1, Solomon2, Cordeau1* and *Cordeau2* between 1 and 4. No optimal solutions are available for those instances mentioned so far. Hence, we will compare our results against the best known solutions so far (BKS).

Additionally, we consider the set of instances proposed in [24], which we will refer to as set *VanSteenwegen*: These instances correspond to the data sets provided in *Solomon1, Solomon2* and *Cordeau1*. For these instances, the number of vehicles was set to the minimum number of vehicles required to serve all customers. Hence, the optimal value of the objective function can be derived in a straightforward way.

Our solution approach was implemented in C++. All experiments where carried out on a Xeon CPU at 3.1 GHz with 32 GB of RAM, which was shared with 7 other CPUs. If not noted otherwise, all reported CPU times are in seconds. Due to the stochastic nature of our algorithm, we report both the best and average solution found within five independent runs.

In the following, we will provide a comprehensive sensitivity analysis with respect to different parameter settings and proposed operators. We then compare the results against the following state-of-the-art algorithms available in the literature: the ant colony (ACO) system developed by [14], the iterated local search (ILS) algorithm proposed in [24], a variable neighborhood search (VNS) by [23], a hybrid evolutionary local search algorithm which has been combined with a greedy randomized adaptive search procedure (GRASP, see [11]), the slow version of the heuristic based on SSA by [12], the LP-based granular variable neighborhood search (GVNS) developed by [10], the iterative three-component heuristic (I3CH) by [9], the iterative local search as presented by [7] and the iterative local search combined with SA as discussed by [6]. Detailed results for the latter are available from the Orienteering Problem Library at `http://centres.smu.edu.sg/larc/orienteering-problem-library/`.

### 4.2 Overall Results

First, we present aggregated results across the various sets of instances. For these instances, our LNS has been parameterized as follows. In every iteration, we destroy up to $d = 40\%$ of the current solution. Nodes to be removed are evaluated according to the resulting reduction in travel time upon removal (*TT-based destroy operator*). We evaluate the consequences upon removal of *sequences* of nodes. Then, the sequence is selected for removal in a *biased* way, the probability of being selected for removal being proportional to the resulting reduction in travel time. In the following repair step, we add nodes depending on a combination of their *squared potential* with a lookahead of $l = 1$ and the corresponding increase in travel time ($POT^2/TT$-based repair operator). The node to be inserted is selected in a *greedy* way, thereby selecting the node with the highest potential (*POT operator).* The run time limit has been set to $N = 100,000$ iterations, and we perform $R = 100$ iterations without improved solutions before we reinitialize the current solution. Justification of all parameter settings is provided in Sect. 4.3. Note that we have investigated combinations of all reasonable parameter settings in a preliminary study to derive the optimal settings.

In Tables 1 and 2, the *set* column denotes the type of instances that have been aggregated. *BKS* reports the average of the best known solutions available so far, *Best* reports the average of the best solution found per instance across the

entire set of instances, and *Avg* gives the average objective function value found by our algorithms. The latter are shown in bold if they meet or are superior to the BKS. $CPU_f$ and $CPU_t$ give the average run times required (in seconds) to find the solution and the run time in total, respectively. Detailed results for each underlying instance can be found in the electronic appendix at `https://goo.gl/jjOvU2`.

For the case of one vehicle (OP, Table 1, $m = 1$), we can find the BKS for all of the *Solomon1* instances and for most of the *Solomon2* instances. Average CPU times are very reasonable: about 1-2 seconds for the *Solomon1* instances, 16-30 seconds for the *Solomon2* instances, 12 seconds for *Cordeau1* and 33 seconds for the *Cordeau2* instances on average. Generally, the *Cordeau* instances are more challenging to solve than the *Solomon* instances. For instances considering two vehicles ($m = 2$), the best solutions obtained by our algorithm are the same as BKS for RC1, C2 and R2. They are quite close for the remaining *Solomon* instances. The average gap is 0.1-0.2% for the *Solomon* and 1.0-2.6% for the *Cordeau* instances. Run times are between about 20 seconds (RC1) and 135 seconds (*Cordeau2*) on average.

Table 1: Aggregated Results Averaged over Various Instance Sets

| set | BKS | Best | Avg | $CPU_f$ | $CPU_t$ | set | BKS | Best | Avg | $CPU_f$ | $CPU_t$ |
|---|---|---|---|---|---|---|---|---|---|---|---|
| *Solomon1, m = 1* | | | | | | *Solomon2* | | | | | |
| C1 | 366.67 | **366.67** | **366.67** | 1.4 | 19.0 | C2 | 932.50 | **932.50** | 931.75 | 3.4 | 47.0 |
| R1 | 281.92 | **281.92** | 281.73 | 2.1 | 15.2 | R2 | 1002.36 | **1003.45** | 1001.00 | 30.8 | 65.6 |
| RC1 | 265.38 | **265.38** | 264.60 | 0.6 | 10.5 | RC2 | 959.25 | 957.38 | 954.88 | 16.0 | 47.2 |
| *Cordeau1* | | | | | | *Cordeau2* | | | | | |
| 1-10 | 462.90 | 462.30 | 461.82 | 11.9 | 40.1 | 11-20 | 534.10 | 525.10 | 521.14 | 33.4 | 67.5 |
| *Solomon1, m = 2* | | | | | | *Solomon2* | | | | | |
| C1 | 673.33 | 672.22 | 672.00 | 5.6 | 30.6 | C2 | 1478.75 | **1478.75** | 1475.75 | 6.4 | 63.5 |
| R1 | 510.83 | 510.08 | 509.27 | 8.0 | 25.5 | R2 | 1410.45 | **1413.55** | 1410.40 | 22.9 | 42.2 |
| RC1 | 510.50 | **510.50** | 510.25 | 3.2 | 20.6 | RC2 | 1566.25 | 1569.25 | 1563.20 | 30.2 | 52.5 |
| *Cordeau1* | | | | | | *Cordeau2* | | | | | |
| 1-10 | 850.00 | 844.60 | 840.12 | 50.1 | 85.0 | 11-20 | 952.70 | 934.70 | 925.04 | 85.9 | 135.1 |
| *Solomon1, m = 3* | | | | | | *Solomon2* | | | | | |
| C1 | 921.11 | 920.00 | 916.44 | 11.5 | 40.5 | C2 | 1810.00 | **1810.00** | **1810.00** | 0.3 | 0.3 |
| R1 | 717.17 | 716.58 | 715.38 | 12.5 | 35.3 | R2 | 1456.45 | 1456.36 | 1456.16 | 1.2 | 4.9 |
| RC1 | 736.25 | 736.13 | 733.58 | 10.9 | 30.2 | RC2 | 1720.13 | 1719.13 | 1718.25 | 5.8 | 14.8 |
| *Cordeau1* | | | | | | *Cordeau2* | | | | | |
| 1-10 | 1159.60 | 1147.80 | 1139.00 | 81.6 | 124.2 | 11-20 | 1271.80 | 1244.30 | 1233.78 | 111.4 | 191.7 |
| *Solomon1, m = 4* | | | | | | *Solomon2* | | | | | |
| C1 | 1136.67 | 1128.89 | 1123.33 | 18.6 | 48.5 | C2 | 1810.00 | **1810.00** | **1810.00** | 0.0 | 0.0 |
| R1 | 892.42 | 891.50 | 889.35 | 18.3 | 42.7 | R2 | 1458.00 | **1458.00** | **1458.00** | 0.0 | 0.0 |
| RC1 | 944.25 | 943.50 | 941.15 | 12.1 | 37.7 | RC2 | 1724.00 | **1724.00** | **1724.00** | 0.0 | 0.1 |
| *Cordeau1* | | | | | | *Cordeau2* | | | | | |
| 1-10 | 1407.10 | 1379.00 | 1366.10 | 92.7 | 156.1 | 11-20 | 1526.00 | 1481.00 | 1467.22 | 137.7 | 234.8 |

With increasing number of vehicles ($m = 3$), we can still determine the BKS for all *Solomon* and C2 instances. For the remaining *Solomon* instances, the gap is negligible. *Cordeau* instances show a gap that is similar to results

with two vehicles. Run times increase for *Solomon1*, *Cordeau1* and *Cordeau2* instances, but decrease heavily for *Solomon2* instances. For instances considering four vehicles ($m = 4$), all BKS of *Solomon2* could be found in a very short run time. The average gap for *Solomon1* remains small (about 0.6%), while the gap for *Cordeau1* and *Cordeau2* increases up to 3.3%. Again, the *Cordeau* instances are harder to solve and require more run time for the same number of iteration.

Table 2 presents the average results for the *VanSteenwegen* instances. Note that for these instances, the number of vehicles was set to the minimum number of vehicles required to service all customers. For the majority of instances, we are able to determine the BKS – except for *Cordeau1* instances, where we observe an average gap of 1.2%.

Table 2: Aggregated Results Averaged over *Vansteenwegen* Instance Sets with Minimum Number of Vehicles

| set | BKS | Best | Avg | $CPU_f$ | $CPU_t$ | set | BKS | Best | Avg | $CPU_f$ | $CPU_t$ |
|---|---|---|---|---|---|---|---|---|---|---|---|
| C1 | 1810.00 | **1810.00** | **1810.00** | 1.50 | 1.52 | C2 | 1810.00 | **1810.00** | **1810.00** | 0.00 | 0.03 |
| R1 | 1449.58 | **1449.58** | 1447.32 | 24.04 | 45.38 | R2 | 1458.00 | **1458.00** | 1457.91 | 5.91 | 9.68 |
| RC1 | 1719.00 | **1719.00** | 1717.28 | 25.34 | 39.08 | RC2 | 1724.00 | **1724.00** | **1724.00** | 2.59 | 2.62 |
| 1-10 | 2270.40 | 2259.00 | 2255.90 | 89.12 | 154.86 | | | | | | |

### 4.3 Parameter Settings

In the following, we investigate and justify the parameter settings obtained for the computation of above results.

**Selection of Destroy and Repair Operators** We proposed five different destroy and repair operators (see Sect. 3.2 and 3.3). They differ in the way consequences upon removal or insertion of nodes are evaluated. We tested all resulting 25 combinations of destroy and repair operators in order to identify the best possible combination for the given benchmark instances (see Table 3). We show average objective function values of the best solutions found (columns "best") as well as averages of all solutions obtained (columns "Avg"). For both cases, the combination of a **TT-based destroy operator** and ***POT*$^2$*/TT*-based repair operator** yields the best overall results.

**Procedure of Selection Operator** When selecting the nodes to be inserted into or removed from the current solution, we tested three variants on how to choose them based on the estimated consequences. Table 4 presents the average results of all combinations of selection procedure of destroy and repair operators. For both average solution values of best solutions obtained and all solutions, the **biased selection** of customer nodes for removal and the **greedy** repair yield the best results.

Table 3: Identifying the Best Combination of Destroy and Repair Operators

| | Best | | | | | Avg | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Destroy / Repair | P | TT | POT | $POT/TT$ | $POT^2/TT$ | P | TT | POT | $POT/TT$ | $POT^2/TT$ |
| Profit (P) | 999.13 | 1001.29 | 1000.50 | 1002.45 | 1002.95 | 995.09 | 997.70 | 998.08 | 999.75 | 1000.36 |
| Travel Time (TT) | 1000.50 | 1001.50 | 1002.09 | 1002.79 | **1003.36** | 997.41 | 997.85 | 1000.32 | 1000.19 | **1001.20** |
| Potential (POT) | 999.00 | 1001.02 | 1000.55 | 1002.73 | 1003.20 | 995.66 | 997.76 | 998.26 | 1000.12 | 1001.07 |
| $POT/TT$ | 999.11 | 1001.61 | 1001.04 | 1002.73 | 1003.09 | 995.94 | 998.10 | 998.51 | 1000.12 | 1000.81 |
| $POT^2/TT$ | 999.20 | 1001.43 | 1000.68 | 1002.79 | 1003.05 | 995.28 | 997.28 | 998.35 | 1000.11 | 1000.48 |

Table 4: Identifying the Best Selection Procedure when Using Destroy and Repair Operators

| | Best | | | Avg | | |
|---|---|---|---|---|---|---|
| Destroy / Repair | Random (R) | Greedy (G) | Biased (B) | Random (R) | Greedy (G) | Biased (B) |
| Random (R) | 987.71 | 1003.04 | 1000.41 | 981.20 | 1000.50 | 996.79 |
| Greedy (G) | 983.25 | 995.27 | 994.63 | 973.58 | 988.11 | 987.08 |
| Biased (B) | 991.43 | **1003.36** | 1001.41 | 985.56 | **1001.20** | 997.84 |

**Degree of Destruction** An important issue for the parameterization of our LNS is the degree of destruction of a solution for further improvement. We investigated the resulting solution quality and run time when varying the degree of destruction $d$ between 10% and 100%. Table 5 shows the corresponding results. As can be observed from the table, CPU times increase significantly with increasing degree of destruction, since this requires increasing efforts of repairing a solution. As can be seen from the table, for the given instances, the **optimal degree of destruction is at 40%**. For this value, the obtained solution quality is best in terms of the average solution quality as well as the average best solution per instance found, while run times still remain at an acceptable level (on average: 12.8 seconds until the solution could be found/38.2 total run time).

Table 5: Effects of Varying $d$ when Destroying Solutions

| $d$ | 10% | 20% | 30% | 40% | 50% | 60% | 70% | 80% | 90% | 100% |
|---|---|---|---|---|---|---|---|---|---|---|
| Best | 1001.04 | 1002.57 | 1002.86 | **1003.36** | 1002.93 | 1003.29 | 1002.86 | 1002.50 | 1002.64 | 1002.32 |
| Avg | 997.26 | 999.89 | 1000.52 | **1001.20** | 1001.11 | 1000.89 | 1001.15 | 1000.58 | 1000.60 | 1000.23 |
| $CPU_f$ | 7.8 | 8.1 | 10.7 | 12.8 | 15.0 | 19.0 | 23.1 | 26.8 | 27.2 | 33.6 |
| $CPU_t$ | 19.3 | 24.6 | 30.6 | 38.2 | 45.0 | 53.0 | 63.9 | 73.0 | 82.5 | 91.6 |

**Length of Removal** As described in Sect. 3.2, we propose to remove sequences of nodes rather than individual nodes. To prove the effectiveness of this approach, we designed the following experiment: Rather than removing subsequences (of dynamic length) of nodes, we also removed them individually. With a slightly superior solution quality than removing single nodes, removing sequences of nodes leads to better results, both in terms of average and best solutions found,

and also reduces the required run time (avg: 1001.20 vs. 999.26; best: 1003.36 vs. 1002.41; $CPU_t$: 38.2 vs. 51.0).

**Lookahead of Potential** For the operator *POT*, we propose to identify the potential of inserting up to $o$ nodes additionally upon destroying or repairing the current solution. In order to identify how many nodes $o$ should be considered, we varied our lookahead parameter between $o = 1$ and $o = 3$. Table 6 shows the corresponding results of this experiment. The variant with $o = 1$ works best. Note that the run time increases significantly when we consider $o > 1$ due to the increasing number of subsets of nodes to be investigated. We have not investigated $o = 0$, since this would correspond to the profit of a single node (and the proposed operator **P**).

Table 6: Effects of Varying Lookahead $o$ of Potential

| $o$ | 1 | 2 | 3 |
|---|---|---|---|
| Best | **1003.36** | 1003.09 | 1001.77 |
| Avg | **1001.20** | 1000.94 | 1000.43 |
| $CPU_f$ | 38.2 | 137.9 | 747.9 |
| $CPU_t$ | 12.8 | 36.3 | 154.6 |

### 4.4 Comparison with other Algorithms

Many related TOPTW papers base their experiments on the same set of instances, which allows for a comparison of the number of BKS obtained by the corresponding algorithm. Table 7 shows the percentage of BKS obtained by most recent TOPTW algorithms at the time they were published compared to the number of BKS our algorithm could obtain. For the cited papers, all *Solomon* and *Cordeau* instances have been considered with a fixed number of vehicles ($m = 1/2/3/4$). We can see from the table that our LNS framework can provide the largest proportion of BKS for the benchmark instances compared with most recent algorithms. Our algorithm seems to be especially beneficial for the instances with two vehicles ($m = 2$). Although our algorithm can provide good solutions in a very short runtime, note that we could not compare the runtimes here due to varying computational environments.

## 5 Conclusion & Outlook

In this paper, we proposed a simple but effective LNS framework including neighborhood operators especially developed for the characteristics of the TOPTW. We presented smart ideas of destroying and repairing solutions, which turned out to be quite effective for the majority of well-known benchmark instances. In particular, we were able to provide 17 new BKS, and our algorithm was able to

Table 7: Percentage of BKS obtained by Different Algorithms, extending [8]

| Reference | Algorithm | Percentage of best known solutions m=1 | m=2 | m=3 | m=4 | Average |
|---|---|---|---|---|---|---|
| Labadie et al. (2011) | GRASP-ELS | 50.0 | 21.1 | 32.9 | 46.1 | 37.5 |
| Lin and Yu (2012) | SSA | 51.3 | 34.2 | 39.5 | 56.6 | 45.4 |
| Labadie et al. (2012) | GVNS | 36.8 | 30.3 | 40.8 | 44.7 | 38.2 |
| Souffriau et al. (2012) | GRILS | 51.3 | 15.8 | 22.4 | 39.5 | 32.3 |
| Hu and Lim (2014) | I3CH | 43.4 | 34.2 | 57.9 | 55.3 | 47.7 |
| Cura (2014) | ABC | 48.7 | 36.8 | 46.1 | 48.7 | 45.1 |
| Gunawan et al. (2015b) | ILS | 68.4 | 51.3 | 56.6 | 55.3 | 57.9 |
| Gunawan et al. (2015a) | SAILS | 67.1 | 50.0 | 57.9 | 53.9 | 57.2 |
| Schmid & Ehmke (2017) | LNS | 82.9 | 71.1 | 65.8 | 60.1 | 70.0 |

provide about 70% of current BKS at reasonable run times. For future research, we would like to consider a variant of the metaheuristic that automatically tunes the parameter $d$, i.e., a framework that automatically balances between diversification and intensification. Furthermore, we think it would be fruitful to extend our solution framework to a parallel version. Considering GPU computing techniques might also be worthwhile in order to improve the efficiency and effectiveness of our LNS. Finally, it would be interesting to compare the performance of the metaheuristic with commercial solvers like CPLEX.

## References

[1] Butt, S.E., Cavalier, T.M.: A heuristic for the multiple tour maximum collection problem. Computers & Operations Research 21, 101–111 (1994)

[2] Chao, I., Golden, B., Wasil, E.: The team orienteering problem. European Journal of Operational Research 88, 475–489 (1996)

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

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

[5] Golden, B., Levy, L., Vohra, R.: The orienteering problem. Naval Research Logistics 34, 307–18 (1987)

[6] Gunawan, A., Lau, H.C., Lu, K.: Sails: hybrid algorithm for the team orienteering problem with time windows. Proceedings of the 7th multidisciplinary international scheduling conference (MISTA 2015), Prague, Czech Republic pp. 276–295 (2015)

[7] Gunawan, A., Lau, H.C., Lu, K.: Well-tuned ils for extended team orienteering problem with time windows. LARC Technical Report Series (2015), http://centres.smu.edu.sg/larc/files/2015/09/Well-Tuned-ILS-for-Extended-Team-Orienteering-Problem-with-Time-WindowsTR-01-15.pdf

[8] Gunawan, A., Lau, H.C., Vansteenwegen, P.: Orienteering problem: A survey of recent variants, solution approaches and applications. European Journal of Operational Research 255(2), 315 – 332 (2016)

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

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

[11] Labadie, N., Melechovsk, J., Wolfler Calvo, R.: Hybridized evolutionary local search algorithm for the team orienteering problem with time windows. Journal of Heuristics 17, 729–753 (2011)

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

[13] Mansini, R., Pelizzari, M., Wolfler Calvo, R.: A granular variable neighborhood search heuristics for the tour orienteering problem with time windows. Technical Report 2008-02-52, Dipartimento di Elettronica per l'Automazione, Università di Brescia (2008)

[14] Montemanni, R., Gambardella, L.: Ant colony system for team orienteering problems with time windows. Foundations of Computing and Decision Sciences 34 (2009)

[15] Pisinger, D., Ropke, S.: Large neighborhood search. In: M, G., J-Y, P. (eds.) Handbook of metaheuristics, pp. 399–419. Springer (2010)

[16] Righini, G., Salani, M.: 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 (2009)

[17] Savelsbergh, M.: The vehicle routing problem with time windows: Minimizing route duration. INFORMS Journal on Computing 4, 146–154 (1992)

[18] Schmid, V.: Hybrid metaheuristic for the team orienteering problem with time windows and service time dependent profits (2014), presentation at VeRoLog 2014, Oslo, Norway

[19] Schmid, V., Gómez Rodríuez, J.S.: On solving routing problems with time windows given dynamic service times and profits (2013), presentation at 26th European conference on operational research, Rome, Italy

[20] Schrimpf, G., Schneider, J., Stamm-Wilbrandt, H., Dueck, G.: Record breaking optimization results using the ruin and recreate principle. Journal of Computational Physics 159, 139–171 (2000)

[21] Solomon, M.: Algorithms for the vehicle routing and scheduling problems with time window constraints. Operations Research 53, 254–265 (1987)

[22] Tang, H., Miller-Hooks, H.: A tabu search heuristic for the team orienteering problem. Computers & Operations Research 32, 1379–1407 (2005)

[23] Tricoire, F., Romauch, M., Doerner, K.F., Hartl, R.F.: Heuristics for the multi-period orienteering problem with multiple time windows. Computers & Operations Research 37, 351–367 (2010)
[24] Vansteenwegen, P., Souffriau, W., Vanden Berghe, G., Van Oudheusden, D.: Iterated local search for the team orienteering problem with time windows. Computers & Operations Research 36, 3281–3290 (2009)