Position Papers of the Federated Conference on Computer Science and Information Systems pp. 71–78
DOI: 10.15439/2015F311
ACSIS, Vol. 6

Time-Dependent Traveling Salesman Problem with Multiple Time Windows

Jarosław Hurkała
Institute of Control & Computation Engineering, Warsaw University of Technology, Warsaw, Poland
Interdisciplinary Center for Security, Reliability and Trust – University of Luxembourg
Email: J.Hurkala@elka.pw.edu.pl

***Abstract*—The TSP, VRP and OP problems with time constraints have one common sub-problem – the task of finding the minimum route duration for a given order of customers. While much work has been done on routing and scheduling problems with time windows, to this date only few articles considered problems with multiple time windows. Moreover, since the assumption of constant travel time between two locations at all times is very unrealistic, problems with time-dependent travel were introduced and studied. Finally, it is also possible to imagine some situations, in which the service time changes during the day. Again, both issues have been investigated only in conjunction with single time windows. In this paper we propose a novel algorithm for computing minimum route duration in traveling salesman problem with multiple time windows and time-dependent travel and service time. The algorithm can be applied to wide range of problems in which a traveler has to visit a set of customers or locations within specified time windows taking into account the traffic and variable service/visit time. Furthermore, we compare three metaheuristics for computing a one-day schedule for this problem, and show that it can be solved very efficiently.**

I. Introduction

In this paper we focus our work on time-dependent routing and scheduling problem with multiple time windows. The problem consist of an agent (tourist, sales representative, etc.) whose aim/duty is to visit a predefined set of customers/locations (e.g. points of interest). Each customer/location may define many time windows which indicates the availability during the day. Furthermore, we assume, that the travel time between the customers/locations changes due to the traffic. In this work we also assume different service/visit time in different time windows.

In this work we describe the theory and algorithms for computing one-day schedule in time-dependent traveling salesman problem with multiple time windows for application in many well known operational research problems such as vehicle routing problem (VRP) (see [2], [3], [8], [14]), orienteering problem (OP) (see [15], [16]), or generally traveling salesman problem (TSP) with complex time constraints. While much work has been done on mixed routing and scheduling problems with time windows, to this date only few articles considered problems with multiple time windows (cf. [2]).

Throughout this article, we will denote a sequence of customers as a route, while we use the term schedule to denote a route with fixed visit times.

The paper is organized as follows: in Section 2 we describe the problem, and discuss additional issues arising from multiple time windows, and time-dependent travel and service time. Section 3 describes preprocessing of time windows and presents the minimum route duration algorithm. In Section 4 we explain in details the algorithms used for computing the one-day schedule in time-dependent traveling salesman problem with multiple time windows. Section 5 shows the results of our numerical experiments. Finally, some concluding remarks are given in Section 6.

II. Problem description

We consider a Time-Dependent Traveling Salesman Problem with Multiple Time Windows (TDTSPMTW) with the following features:

1) each customer can define multiple time windows during which he is available and can be serviced;
2) the service time can be different in every time window of the customer;
3) the travel time depends on the traffic time zone, in which the transit actually occurs;
4) starting and ending depots are treated as customers so that they also have time windows.

The TDTSPMTW problem can be defined as follows.

A. Problem notation

Let $\mathcal{I} = \{1, \ldots, n\}$ be the set of customers $i \in \mathcal{I}$ that are visited by the traveling salesman. Let $\mathcal{W}_i$, be the set of i-th customer time windows $j \in \mathcal{W}_i$, during which the visit can take place. The set of time windows of all the customers will be denoted by $\mathcal{W} = \bigcup_{i \in \mathcal{I}} \mathcal{W}_i$. Thus, $[a_i^j, b_i^j]$ will denote the j-th time window of i-th customer, where a_i^j is the beginning, and b_i^j is the end of the time window, and the service time of i-th customer in j-th time window will be denoted by s_i^j. Let $\mathcal{Z}$, $k \in \mathcal{Z}$, be the set of traffic time zones $[p^k, q^k]$, where p^k is the beginning, and q^k is the end of the traffic time zone, and t_i^k be the travel time from i-th customer to the next one in the sequence, in k-th time zone. *Notice, that we deliberately define travel to the next customer instead of to the current one - this significantly simplifies the considerations*. The visit at i-th customer will be denoted by $[\alpha_i, \beta_i]$, where α_i is the time of arrival at the customer, and β_i is the departure time of the visit.

B. Traffic time zones

Before we can proceed with explaining the minimum route duration algorithm, the problem of traffic time zones has to be accommodated. Since the customers can already have multiple time windows, we can take advantage of this property and create additional, „virtual“ time windows so that the travel time in each window is well-defined. For every time window we have to check whether it lies in one traffic time zone, or maybe spans across multiple zones. In the latter case, the original time window has to be divided into smaller, overlapping windows. We shall explain this on the following example. Let $[a, b]$ be a time window with service time s, that spreads over two time zones: $[p^1, q^1]$ with travel time t^1 and $[p^2, q^2]$ with travel time t^2, such that $p^1 < a < q^1 = p^2 < b < q^2$. Than, we need to divide the original window into the following ones: $[a, q^1]$ with service time s and travel time t^1 and $[p^2 - s, b]$ with service time s and travel time t^2. Notice, that the beginning of the second window is brought forward by the service time, because we consider travel time to the next customer in sequence, and the transit starts as soon as the current customer has been serviced. Thus, let $\mathcal{V} = \bigcup_{i \in \mathcal{I}} \mathcal{V}_i$ be the set of virtual time windows of customers $i \in \mathcal{I}$. Finally, we can define a function $\gamma : \mathcal{V} \to \mathcal{Z}$ that maps the virtual time windows into the time zones. Since each virtual time window falls within one traffic time zone, we know that the function is well-defined.

C. The normalized formulation

Having the travel time unambiguously defined for every time window we can normalize the model similarly to [8], [16]. Thus, we merge the service time and the travel time into one parameter $d_i^j = s_i^j + t_i^{\gamma(j)}$, denoting the visit duration. At the same time, we postpone the ending of every (virtual) time window by the travel time associated with it, i.e. windows $[a_i^j, b_i^j]$ are transformed into $[a_i^j, b_i^j + t_i^{\gamma(j)}]$. *Notice, that from this point on in the article the departure time will have new meaning, i.e., the moment the salesman reaches the next customer in the sequence.*

D. Master problem

Let $\pi = (\pi(1), \pi(2), \ldots, \pi(n))$ be the permutation of customers.

The master problem of TDTSPMTW, i.e. the problem of finding optimal sequence of customers to be visited during one day, can be defined as follows:

$$\pi^* = \arg\min_{\pi} MinimumRouteDuration(\pi) \tag{1}$$

The problem of finding the minimum route duration for a given sequence of customers (i.e. the subproblem of TDTSPMTW) can be formulated as follows.

E. Subproblem formulation

Let us define the main decision variables and explain their meaning. For the time windows selection we define:

$$y_i^j = \begin{cases} 1 & \text{if the visit at customer i takes place} \\ & \text{within the time window j} \\ 0 & \text{else} \end{cases} \tag{2}$$

Using these notations, we write a mathematical model for the TDTSPMTW subproblem. The objective is to minimize the route duration (calculated as a difference between departure time of last customer and arrival time at first customer):

$$\min \beta_n - \alpha_1 \tag{3}$$

$$(a_i^j + d_i^j) y_i^j \le \beta_i \le b_i^j + \mathcal{M}(1 - y_i^j) \quad i \in \mathcal{I}, j \in \mathcal{V} \tag{4}$$

$$\sum_{j \in \mathcal{V}_i} y_i^j = 1 \quad i \in \mathcal{I} \tag{5}$$

$$\alpha_i = \beta_i - \sum_{j \in \mathcal{V}_i} d_i^j y_i^j \quad i \in \mathcal{I} \tag{6}$$

$$\beta_i \le \alpha_{i+1} \quad i \in \mathcal{I} \setminus \{n\} \tag{7}$$

$$y \in \{0, 1\}, \beta \ge 0 \tag{8}$$

Constraints (4) handle the time windows in a classical way (the departure time must be within a time window) with the noticeable addition of the upper index, since we have multiple time windows. Constraints (5) ensure that exactly one time window per customer is chosen. The auxiliary constraints (6) compute the start of every visit (arrival time α). Finally, constraints (7) forbid to start the next visit before the current customer departure time, so that the visits do not overlap. The y variables are binary, and departure times β are non-negative real variables (8).

III. Minimum route duration algorithm

The minimum route duration algorithm that we have developed requires a feasible solutions to start with. Hence, the preprocessing of the virtual time windows is needed.

A. Preprocessing

In order for the algorithm to work, unnecessary time windows from the bottom-right (see Algorithm 1) and top-left (see Algorithm 2) corners have to be removed (if you imagine the first window of the first customer in the bottom-left corner of a diagram, and the last window of the last customer in the top-right corner). Similar approach has already been proposed in [16], but in our formulation the visit duration may be different in subsequent time windows of a customer, hence we can dismiss only the ones that lead to unfeasible schedule.

Algorithm 1 TDTSPMTW: top-down preprocessing

```
α ← max_{j∈V_n} {b_n^j − d_n^j}
for i = n − 1 to 1 do
    V_i ← V_i \ {j ∈ V_i : a_i^j + d_i^j > α}
    if V_i = {∅} then
        return false {schedule not feasible}
    end if
    for j ∈ V_i do
        b_i^j ← min{b_i^j, α}
    end for
    α ← max_{j∈V_i} {b_i^j − d_i^j}
end for
return true
```

Algorithm 2 TDTSPMTW: bottom-up preprocessing

1: $\beta \leftarrow \min_{j \in \mathcal{V}_1}\{a_1^j + d_1^j\}$
2: **for** $i = 2$ **to** n **do**
3: $\quad \mathcal{V}_i \leftarrow \mathcal{V}_i \setminus \{j \in \mathcal{V}_i : b_i^j - d_i^j < \beta\}$
4: $\quad$ **if** $\mathcal{V}_i = \{\emptyset\}$ **then**
5: $\quad\quad$ **return false** {schedule not feasible}
6: $\quad$ **end if**
7: $\quad$ **for** $j \in \mathcal{V}_i$ **do**
8: $\quad\quad a_i^j \leftarrow \max\{a_i^j, \beta\}$
9: $\quad$ **end for**
10: $\quad \beta \leftarrow \min_{j \in \mathcal{V}_i}\{a_i^j + d_i^j\}$
11: **end for**
12: **return true**

Algorithm 3 TDTSPMTW: constructing feasible sub-schedule

Require: i, β
1: **while** $i \leq n$ **do**
2: $\quad \beta \leftarrow \min_{j \in \mathcal{V}_i, b_i^j - d_i^j \geq \beta} \max\{a_i^j, \beta\} + d_i^j$
3: $\quad i \leftarrow i + 1$
4: **end while**
5: **return** β

Algorithm 4 TDTSPMTW: constructing dominant sub-schedule

Require: i, α
1: **while** $i \geq 1$ **do**
2: $\quad \alpha \leftarrow \max_{j \in \mathcal{V}_i, a_i^j + d_i^j \leq \alpha} \min\{b_i^j, \alpha\} - d_i^j$
3: $\quad i \leftarrow i - 1$
4: **end while**
5: **return** α

Algorithm 5 TDTSPMTW: minimum route duration

1: $\tau^* \leftarrow \infty$
2: **if** TopDownPreprocessing() **and** BottomUpPreprocessing() **then**
3: $\quad$ **for** $i = 1$ **to** n **do**
4: $\quad\quad$ **for** $j \in \mathcal{V}_i$ **do**
5: $\quad\quad\quad \beta \leftarrow constructFeasibleSubSchedule(i + 1, b_i^j)$
6: $\quad\quad\quad \alpha \leftarrow constructDominantSubSchedule(i - 1, b_i^j - d_i^j)$
7: $\quad\quad\quad \tau \leftarrow \beta - \alpha$
8: $\quad\quad\quad$ **if** $\tau < \tau^*$ **then**
9: $\quad\quad\quad\quad \tau^* \leftarrow \tau$
10: $\quad\quad\quad$ **end if**
11: $\quad\quad$ **end for**
12: $\quad$ **end for**
13: **end if**
14: **return** τ^*

The top-down preprocessing algorithm (Algorithm 1) begins with calculating the latest possible arriving at the last customer (denoted by α). Than, starting from the second to last customer, virtual time windows are removed, for which the visit starting at the beginning of the window would exceed α (line 3). If after this step there is no more time windows, the schedule (sequence of customers) is not feasible, and the algorithm terminates. Otherwise, the remaining windows are tightened so that the ending of each window does not exceed α (line 8). Finally, the α value is recalculated for the current customer (line 10) and the procedure starts over with previous customer.

The bottom-up preprocessing algorithm (Algorithm 2) works almost identically as top-down. The differences are as follows:

- the earliest possible departure time from the customer (denoted by β) is taken into consideration (lines: 1, 10);
- algorithm iterates from the second customer to the last one;
- windows are removed if the beginning of visit that starts as late as possible overlaps the departure time β (line 3);
- the remaining windows are tightened so that they do not begin earlier than β (line 8).

The minimum route duration algorithm that we propose in this paper consists of iteratively reviewing schedules of which the one with the shortest duration is chosen. The procedure of constructing each schedule is divided into two phases.

B. Phase 1: constructing feasible sub-schedule

The first phase consist of computing a feasible sub-schedule that starts from a given customer, and ends with the last one (see Algorithm 3). This procedure requires an index of a customer and an initial departure time. Repeatedly, among the virtual time windows that have enough room to fit the visit after the given departure time ($b_i^j - d_i^j \geq \beta$), the earliest visit departure time (calculated as: $\max\{a_i^j, \beta\} + d_i^j$) is searched for.

C. Phase 2: constructing dominant sub-schedule

This phase is based on the notion of the dominant solutions (see [8], [14], [16]). To put it simply, a schedule with starting time α^1 dominates schedule with starting time α^2, if $\alpha^1 > \alpha^2$ and at the same time ending time $\beta^1 = \beta^2$ (cf. [8]). In our procedure, instead of fixed visit departure time, we use the arrival time (obviously, $\alpha = \beta - d$). Nevertheless, the principle is the same - we repeatedly search for the latest possible starting time of a visit (calculated as: $\min\{b_i^j, \alpha\} - d_i^j$) among the virtual time windows, that are suitable ($a_i^j + d_i^j \leq \alpha$), i.e. windows in which the currently considered visit does not overlap the initial one.

D. The main algorithm

The algorithm enumerates schedules, constructing them in a particular fashion. For each virtual time window, first, a feasible sub-schedule is constructed with the initial departure time set to the **end** of the current time window (line 5). Secondly, a dominant sub-schedule is constructed with the initial arrival time set to the latest possible start of the visit in the considered window (line 6). The route duration is than computed as the difference between the departure time from

last customer and arrival time at the first customer (line 7). The best solution found during the process is stored (lines 8–10) and returned (line 14). For the algorithm overview see Algorithm 5.

Although the main procedure of the algorithm looks self-explanatory, the reason it finds the optimal solution is not trivial. Savelsbergh [14] has introduced the concept of forward time slack to postpone the beginning of service at a given customer. It can be proven, that the optimal schedule can be postponed until one of the visits **ends** with the time window. Hence, by iteratively reviewing schedules one by one so that every possible time window with visit at the end of it is taken into consideration, an optimal schedule is found by our algorithm.

E. Computational complexity

The preprocessing procedures have both $O(|\mathcal{V}|)$ time complexity. The sub-schedule construction procedures have together $O(|\mathcal{V}|)$ time complexity (they consider disjoint sets of time windows). The main procedure has $O(|\mathcal{V}|)$ time complexity (every time window is taken into consideration). Hence, the total time complexity of the algorithm is $O(|\mathcal{V}|^2)$.

IV. TDTSPMTW MASTER PROBLEM ALGORITHMS

For solving the TDTSPMTW master problem, i.e. computing the one-day schedule, we have chosen three different metaheuristics.

A. Simulated annealing

Simulated annealing was first introduced by Kirkpatrick [10], while Černý [1] pointed out the analogy between the annealing process of solids and solving combinatorial problems. Researchers have been studying the application of the SA algorithm in various fields of optimization. Koulamas [11] presented a survey of operational research problems in which the heuristic was applied. The effectives of the algorithm was also inspected in particular by Hurkała and Hurkała [5], [6], and also Hurkała and Śliwiński [7].

The optimization process of the simulated annealing algorithm can be described in the following steps. Before the algorithm can start, an initial solution is required. Then, repeatedly, a candidate solution is randomly chosen from the neighborhood of the current solution. If the candidate solution is the same or better than the current one, it is accepted and replaces the current solution. A worse solution than the current one still has a chance to be accepted with, so called, acceptance probability. This probability is a function of difference between objective values of both solutions and depends on a control parameter taken from the thermodynamics, called temperature. The temperature is decreased after a number of iterations, and the process continues as described above. The optimization is stopped either after a maximum number of iterations or when a minimum temperature is reached. The best solution found during the annealing process is considered final. For the algorithm overview see Algorithm 6.

Algorithm 6 Simulated annealing

Require: Initial solution s_1
1: $s^* \leftarrow s_1$
2: **for** $i = 1$ **to** N **do**
3: **for** $t = 1$ **to** N_{const} **do**
4: $s_2 \leftarrow perturbate(s_1)$
5: $\delta \leftarrow C(s_2) - C(s_1)$
6: **if** $\delta \leq 0$ or $e^{-\delta/k\tau} > random(0,1)$ **then**
7: $s_1 \leftarrow s_2$
8: **end if**
9: **if** $C(s_2) < C(s^*)$ **then**
10: $s^* \leftarrow s_2$
11: **end if**
12: **end for**
13: $\tau \leftarrow \tau * \alpha$
14: **end for**
15: **return** s^*

The main building block of the simulated annealing is the temperature decrease (also known as the cooling process), which consists of decreasing the temperature by a reduce factor. The parameters associated with this mechanism are as follows:

1) Initial temperature.
2) Function of temperature decrease in consecutive iterations.
3) The number of iterations at each temperature (Metropolis equilibrium).
4) Minimum temperature at which the algorithm terminates or alternatively the maximum number of iterations as the stopping criterion.

Let τ be the temperature and α be the reduce factor. Then the annealing scheme can be represented as the following recursive function:

$$\tau^{i+1} = \alpha * \tau^i, \tag{9}$$

where i is the number of current iteration in which the cooling schedule takes place.

Second building block of SA that has to be customized for a particular problem is the acceptance probability function, which determines whether to accept or reject candidate solution that is worse than the current one. The most widely used function is the following:

$$p\left(\delta, \tau\right) = e^{-\delta/k\tau}, \tag{10}$$

where $\delta = E(s_2) - E(s_1)$ is the difference between the objective value (denoted by E) of the candidate (s_2) and the current solution (s_1), and k is the Boltzmann constant found by:

$$k = \frac{\delta^0}{\log \frac{p^0}{\tau^0}}, \tag{11}$$

where δ^0 is an estimated difference between objective values of two solutions, p^0 is the initial value of the acceptance probability and τ^0 is the initial temperature. Notice that we use

decimal logarithm rather than natural, which is most widely seen in the literature.

B. List-based threshold accepting

List-based threshold accepting algorithm (LBTA) introduced by Lee [12], [13] is an extent of the threshold accepting meta-heuristic, which belongs to the randomized search class of algorithms. The search trajectory crosses the solution space by moving from one solution to a random neighbor of that solution, and so on. Unlike the greedy local search methods which consist of choosing a better solution from the neighborhood of the current solution until such can be found (hill climbing), the threshold accepting allows choosing a worse candidate solution based on a threshold value. In the general concept of the threshold accepting algorithm it is assumed that a set of decreasing threshold values is given before the computation or an initial threshold value and a decrease schedule is specified. The rate at which the values decrease controls the trade-off between diversification (associated with large threshold values) and intensification (small threshold values) of the search. It is immensely difficult to predict how the algorithm will behave when a certain decrease rate is applied for a given problem without running the actual computation. It is also very common that the algorithm with the same parameters works better for some problem instances and significantly worse for others. These reflections led to the list-based threshold accepting branch of threshold accepting meta-heuristic.

In the list-based threshold accepting approach, instead of a predefined set of values, a list is dynamically created during a presolve phase of the algorithm. The list, which in a way contains knowledge about the search space of the underlying problem, is then used to solve it.

The first phase of the algorithm consists of gathering information about the search space of the problem that is to be solved. From an initial solution a neighbor solution is created using a move function (perturbation operator) chosen at random from a predefined set of functions. If the candidate solution is better than the current one, it is accepted and becomes the current solution. Otherwise, a threshold value is calculated as a relative change between the two solutions:

$$\Delta = (C(s_2) - C(s_1))/C(s_1) \tag{12}$$

and added to the list, where $C(s_i)$ is the objective function value of the solution $s_i \in S$, and S is a set of all feasible solutions. For this formula to work, it is silently assumed that $C : S \to \mathbb{R}_+ \cup \{0\}$. This procedure is repeated until the specified size of the list is reached. For the algorithm overview see Algorithm 7.

The second phase of the algorithm is the main optimization routine, in which a solution to the problem is found. The algorithm itself is very similar to that of the previous phase. We start from an initial solution, create new solution from the neighborhood of current one using one of the move function, and compare both solutions. If the candidate solution is better, it becomes the current one. Otherwise a relative change is calculated. To this point algorithms in both phases are identical. The difference in the optimization procedure is that we compare the threshold value with the largest value from the list. If the new threshold value is larger, then the new solution is discarded. Otherwise, the new threshold value replaces the value from the list, and the candidate solution is accepted to next iteration. The best solution found during the optimization process is considered final.

Algorithm 7 Creating the list of threshold values

Require: Initial solution s_1, list size S, set of move operators $m \in M$

```
i ← 0
while i < N do
    m ← random(M)
    s2 ← m(s1)
    if C(s1) ≤ C(s2) then
        Δ ← (C(s2) − C(s1))/C(s1)
        list ← list ∪ {Δ}
        i ← i + 1
    else
        s1 ← s2
    end if
end while
return list
```

The list-based threshold accepting algorithm also incorporates early termination mechanism: after a (specified) number of candidate solutions is subsequently discarded, the optimization is stopped, and the best solution found so far is returned. The optimization procedure of the list-based threshold accepting algorithm is shown in Algorithm 8.

The original LBTA algorithm does not have a solution space independent stopping criterion. If the number of subsequently discarded worse solutions is set too high, the algorithm will run for an unacceptable long time (it has been observed during preliminary tests). Hence, we propose to use additionally a global counter of iterations so that when a limit is reached, the algorithm terminates gracefully.

In the first phase of the list-based threshold accepting algorithm the list is populated with values of relative change between two solutions $\Delta \geq 0$. After careful consideration, we believe that including zeros in the list is a misconception. In the actual optimization procedure, i.e. the second phase, the threshold value is computed only if the new solution is worse than the current one, which means that the calculated relative change will always have a positive value ($\Delta_{new} > 0$). The new threshold value is compared with the largest value from the list (T_{hmax}). Thus, we can distinguish three cases:

1) $T_{hmax} = 0$: since thresholds are non-negative from definition, in this case the list contains all zero elements and it will not change throughout the whole procedure (T_{hmax} is constant). Comparing a positive threshold value Δ_{new} against zero yields in discarding the candidate solution. The conclusions are as follows:
    a) it does not matter how many zeros are in the list,

Algorithm 8 LBTA optimization procedure

Require: Initial solution s_1, thresholds list L, set of move operators $m \in M$

$i \leftarrow 0$
$s^* \leftarrow s_1$
while $i \leq N$ **do**
  $m \leftarrow random(M)$
  $s_2 \leftarrow m(s_1)$
  $i \leftarrow i + 1$
  if $C(s_2) \leq C(s_1)$ **then**
    if $C(s_2) \leq C(s^*)$ **then**
      $s^* \leftarrow s_2$
    end if
    $s_1 \leftarrow s_2$
    $i = 0$
  else
    $\Delta_{new} \leftarrow (C(s_2) - C(s_1))/C(s_1)$
    if $\Delta_{new} < max(list)$ **then**
      $list \leftarrow list \setminus \{max(list)\}$
      $list \leftarrow list \cup \{\Delta_{new}\}$
      $s_1 \leftarrow s_2$
      $i = 0$
    end if
  end if
end while
return s^*

the effective size of the list is equal to one,

b) the algorithm is reduced to hill climbing algorithm that accepts candidate solutions which are at least as good as the current one.

2) $T_{hmax} > 0$ and $\Delta_{new} < T_{hmax}$: the largest (positive) threshold value from the list T_{hmax} is replaced by a smaller (positive) threshold value Δ_{new}. The number of zero elements in the list remains the same throughout the whole procedure and therefore is completely irrelevant to the optimization process. The effective list size is equal to the number of positive elements.
3) $T_{hmax} > 0$ and $\Delta_{new} \geq T_{hmax}$: the new solution is discarded and the list remains unchanged.

The main idea behind the list is to control the diversification and intensification of the search process. In the early stage of the search the algorithm should allow to cover as much solution space as possible, which means that the thresholds in the list are expected to be large enough to make that happen. In the middle stage, the algorithm should slowly stop fostering the diversification and begin to foster the intensification of the search. In the end stage, the intensification should be the strongest, i.e. the list is supposed to contain smaller and smaller threshold values, which induces discarding of worse solution candidates. As a consequence, the algorithm is converging to a local or possibly even a global optimum.

Algorithm 9 Variable neighborhood descent

Require: Initial solution s_0

$s_1 \leftarrow s_0$
$i \leftarrow 1$
repeat
  for $j = 1$ **to** $size(N_i)$ **do**
    $s_2 \leftarrow N_i(s_0)$
    if $C(s_2) < C(s_1)$ **then**
      $s_1 \leftarrow s_2$
    end if
  end for
  if $C(s_1) < C(s_0)$ **then**
    $s_0 \leftarrow s_1$
    $i \leftarrow 1$
  else
    $i \leftarrow i + 1$
  end if
until $i = |N|$
return s_0

Algorithm 10 Reduced variable neighborhood search

Require: Initial solution s_0

$i \leftarrow 1$
repeat
  $s_1 \leftarrow VND(N_i(s_0))$
  if $C(s_1) < C(s_0)$ **then**
    $s_0 \leftarrow s_1$
    $i \leftarrow 1$
  else
    $i \leftarrow i + 1$
  end if
until $i = |N|$
return s_0

C. Variable neighborhood search

Variable neighborhood search (VNS) is a metaheuristic algorithm proposed by Mledanović [9]. This global optimization method is based on an idea of systematically changing the neighborhood in the descent to local minima and in the escape from the valleys which contain them. It has already been successfully used in different Vehicle Routing Problems.

The VNS algorithm consists of two building blocks: variable neighborhood descent (VND) and reduced variable neighborhood search (RVNS).

The optimization process of VND can be explained as follows. First, an initial solution is required. Within a given neighborhood a candidate solution is repeatedly generated and it replaces the current one if it is better. After a specified number of iterations (neighborhood size) the neighborhood is changed to the first one if a better than initial solution has been found. Furthermore, the best solution found so far replaces the initial solution. Otherwise, if the search resulted in no better solutions that the initial one, the current neighborhood is changed to the next one. Either way, the whole operation is

repeated again until the search gets stuck in a local optimum. The best solution found during this process is returned. For the overview of the algorithm see Algorithm 9.

The RVNS is a stochastic algorithm that executes the VND with different initial solutions. This simple procedure, which in fact is quite similar to the VND, can be described in the following few steps. First, an initial solution - a starting point - is required. Within a given neighborhood a candidate solution is repeatedly generated from the initial one, and passed to the VND procedure. If the VND returns a solution that is better than the current one, it gets replaced, and the algorithm starts over from the first neighborhood. Otherwise, the neighborhood is changed to the next one. The whole process can be repeated until the stopping criterion (e.g. specified number of evaluations, time limit) is met. The optimization procedure of RVNS is shown in Algorithm 10.

D. Neighborhood function

The most problem-specific mechanism of SA, LBTA and VNS algorithms, that always needs a different approach and implementation, is the procedure of generating a candidate solution from the neighborhood of the current one, which is called a perturbation scheme, transition operation/operator or a move function. Although there are many ways to accomplish this task for the traveling salesman problem, we have chosen the following three operators:

1) interchanging two adjacent customers,
2) interchanging two random customers,
3) moving a single, random customer to a randomly chosen position.

V. Numerical experiments

The numerical experiments were performed on a number of randomly generated problem instances of different size. The algorithms were implemented in C++. All the computations were executed on the Intel Core i7 3.4GHz microprocessor.

To better compare relative performance of the three algorithms, the only stopping criterion for single run was reaching the same number of schedule evaluations for all computations and problem sizes. This way we could compare the speed as well as the convergence per iteration.

The resulting one-day schedules are presented in Table I. The first column indicates the instance number. The number of customers (second column) ranges from 13 to 23. The route durations found by the three algorithms are shown in columns 3-5. The values in columns 6-8 indicate relative difference between the algorithms outcomes. In order of brevity, we show the computation time in one (the last) column for the given problem instance - it was almost the same for every algorithm due to the (identical) stopping criterion.

The algorithms produced similar results in terms of both the solution quality and the computation time. For some instances one algorithm produces better results, while for some other it is the other way around. Generally, the VNS tends to find a little bit better solutions: 2.49% on average than LBTA, and 2.97% than SA. LBTA and SA are on the other hand almost identical - the former is on average better by only 0.49% than the latter.

VI. Conclusions

We have developed a novel and efficient algorithm that computes the minimum route duration for the Time-Dependent Traveling Salesman Problem with Multiple Time Windows, and compared three metaheuristic algorithms that computes the one-day schedule, which can be successfully utilized in time-oriented TSP, VRP, OP, and other mixed routing and scheduling problems. The minimum route duration algorithm guarantees finding optimal solution in quadratic time in terms of the total number of time windows. To our knowledge, this is the first attempt of solving this kind of time-dependent TSP with multiple time windows.

Acknowledgment

This research was financed by the European Union through the European Regional Development Fund under the Operational Programme “Innovative Economy” for the years 2007-2013; Priority 1 – Research and development of modern technologies under the project POIG.01.03.01-14-076/12: “Decision Support System for Large-Scale Periodic Vehicle Routing and Scheduling Problems with Complex Constraints” and has been supported by the European Union in the framework of European Social Fund through the project: Supporting Educational Initiatives of the Warsaw University of Technology in Teaching and Skill Improvement Training in the Area of Teleinformatics.

References

[1] Černý, V., Thermodynamical approach to traveling salesman problem: An efficient simulation algorithm. J. Optim. Theory Appl., vol. 45 (1985) 41–51.

[2] Belhaiza S., P. Hansen, G. Laporte. *A hybrid variable neighborhood tabu search heuristic for the vehicle routing problem with multiple time windows*, Computers & Operations Research, Volume 52 (2014), 269-281.

[3] Favaretto D., E. Moretti, P. Pellegrini. *Ant colony system for a VRP with multiple time windows and multiple visits*, Journal of Interdisciplinary Mathematics, Volume 10, Issue 2 (2007), 263-284.

[4] Hurkała J., *Minimum Route Duration Algorithm for Traveling Salesman Problem with Multiple Time Windows*, Vehicle Routing and Logistics Optimization, June 8-10, 2015, Vienna, Austria [accepted for presentation].

[5] Hurkała, J. and Hurkała, A., Effective Design of the Simulated Annealing Algorithm for the Flowshop Problem with Minimum Makespan Criterion, Journal of Telecommunications and Information Technology 2 (2012) 92–98.

[6] Hurkała, J. and Hurkała, A., Fair optimization with advanced aggregation operators in a multicriteria facility layout problem, in Proceedings of the 2013 Federated Conference on Computer Science and Information Systems, IEEE, 2013, 355–362.

[7] Hurkała, J. and Śliwiński, T., Fair flow optimization with advanced aggregation operators in Wireless Mesh Networks, in Proceedings of the Federated Conference on Computer Science and Information Systems, IEEE, 2012, 415–421.

[8] Jong C., G. Kant, A. van Vliet. *On Finding Minimal Route Duration in the Vehicle Routing Problem with Multiple Time Windows*, Tech. Rep., The Netherlands: Department of Computer Science, Utrecht University, 1996.

[9] Mladenović, N., and Hansen, P. *Variable neighborhood search*. Computers and Operations Research 24 (11), 1997, 1097-?1100.

[10] Kirkpatrick, S., Gellat, C.D. and Vecchi, M.P., Optimization by simulated annealing, Science, vol. 220 (1983) 671–680.

TABLE I
ROUTE DURATIONS AND COMPUTATION TIME

#	$\|\mathcal{I}\|$	VNS	LBTA	SA	$VNS/LBTA$	$VNS/LBTA$	$LBTA/SA$	Time [s]
1	13	28113	28532	29608	-1.47%	-5.05%	-3.63%	0.281
2	13	27921	27921	29793	0.00%	-6.28%	-6.28%	0.314
3	13	31193	31193	31193	0.00%	0.00%	0.00%	0.283
4	13	29913	31540	31591	-5.16%	-5.31%	-0.16%	0.293
5	13	28113	28532	29608	-1.47%	-5.05%	-3.63%	0.224
6	13	27921	27921	29793	0.00%	-6.28%	-6.28%	0.264
7	14	28314	27299	28289	3.72%	0.09%	-3.50%	0.299
8	15	28134	28039	28325	0.34%	-0.67%	-1.01%	0.381
9	15	28134	28039	28325	0.34%	-0.67%	-1.01%	0.243
10	16	30143	31400	30442	-4.00%	-0.98%	3.15%	0.376
11	16	27822	30210	30078	-7.90%	-7.50%	0.44%	0.282
12	16	35438	35438	35438	0.00%	0.00%	0.00%	0.405
13	16	29957	31453	29843	-4.76%	0.38%	5.39%	0.298
14	17	32442	33306	33306	-2.59%	-2.59%	0.00%	0.499
15	17	33609	33609	33609	0.00%	0.00%	0.00%	0.444
16	17	31585	31730	32515	-0.46%	-2.86%	-2.41%	0.285
17	17	30691	30450	31648	0.79%	-3.02%	-3.79%	0.279
18	17	29136	29136	30553	0.00%	-4.64%	-4.64%	0.295
19	17	31882	34707	34707	-8.14%	-8.14%	0.00%	0.401
20	18	36246	37569	36759	-3.52%	-1.40%	2.20%	0.517
21	18	36389	38696	38696	-5.96%	-5.96%	0.00%	0.399
22	19	40102	40102	40102	0.00%	0.00%	0.00%	0.390
23	19	36231	36231	36231	0.00%	0.00%	0.00%	0.388
24	20	37982	40187	39746	-5.49%	-4.44%	1.11%	0.571
25	20	38107	39943	39897	-4.60%	-4.49%	0.12%	0.360
26	20	38511	38195	38163	0.83%	0.91%	0.08%	0.525
27	20	38081	35533	38577	7.17%	-1.29%	-7.89%	0.470
28	20	40163	40707	40707	-1.34%	-1.34%	0.00%	0.341
29	20	37977	38930	38930	-2.45%	-2.45%	0.00%	0.303
30	21	38468	40434	40379	-4.86%	-4.73%	0.14%	0.510
31	21	39072	39855	39711	-1.96%	-1.61%	0.36%	0.449
32	21	36876	40170	38456	-8.20%	-4.11%	4.46%	0.423
33	21	34468	37702	37702	-8.58%	-8.58%	0.00%	0.376
34	21	39069	40912	40912	-4.50%	-4.50%	0.00%	0.363
35	21	39857	40497	40497	-1.58%	-1.58%	0.00%	0.347
36	22	38856	40044	39298	-2.97%	-1.12%	1.90%	0.550
37	22	37854	40243	40243	-5.94%	-5.94%	0.00%	0.554
38	22	40608	40608	40608	0.00%	0.00%	0.00%	0.557
39	22	40017	40017	40017	0.00%	0.00%	0.00%	0.522
40	22	39997	39997	39997	0.00%	0.00%	0.00%	0.495
41	22	38292	40830	40830	-6.22%	-6.22%	0.00%	0.380
42	22	36783	40711	39449	-9.65%	-6.76%	3.20%	0.446
43	23	36057	36057	37616	0.00%	-4.14%	-4.14%	0.725
44	23	37725	40688	40688	-7.28%	-7.28%	0.00%	0.533
45	23	38952	40366	39465	-3.50%	-1.30%	2.28%	0.538

[11] Koulamas, C., Antony, S.R. and Jaen, R., A survey of simulated annealing applications to operations research problems, Omega, 22 (1994) 41–56.

[12] Lee, D.S., Vassiliadis, V.S., Park, J.M., A novel thresholdaccepting metaheuristic for the job-shop scheduling problem. Computers & Operations Research, 31 (2004) 2199–2213.

[13] Lee, D.S., Vassiliadis, V.S., Park, J.M., List-Based Threshold-Accepting Algorithm for Zero-Wait Scheduling of Multiproduct Batch Plants, Ind. Eng. Chem. Res. 41 (25), 2002, pp. 6579-?6588.

[14] Savelsbergh, M.W.P. *The Vehicle Routing Problem with Time Windows: Minimizing Route Duration*, ORSA Journal on Computing, Vol. 4, Issue 2 (1992), 146-161.

[15] Souffriau W., P. Vansteenwegen, G.V. Berghe, D. Van Oudheusden. . *The Multi-Constraint Team Orienteering Problem with Multiple Time Windows*, Transportation Science, Volume 47, Issue 1 (2013), 53-63.

[16] 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 (2010), 351-367.